ILOG CPLEX 8.1

Advanced Reference Manual

December 2002

Printed in France

Table of Contents

About This Manual

This manual documents advanced routines of the ILOG CPLEX Component Libraries, version 8.1. It supplements the *ILOG CPLEX Reference Manual*. For Callable Library routines such as `CPXgetcallbackinfo()`, mentioned but not documented in this manual, consult the *ILOG CPLEX Reference Manual*, available on-line in the standard distribution of the product.

The examples mentioned in this document are available by anonymous login at the CPLEX ftp site, `ftp://ftp.cplex.com/pub/examples/v81/advanced/src`, in the subdirectory corresponding to the version number of the product.

The chapters in Part I, *Concepts*, provide background information on using the advanced presolve and MIP control routines, as well as information on user cuts and lazy constraints, and should be read before using those features.

All of the advanced routines are documented in alphabetical order in Part II, *Routines*, of this manual. They serve a variety of purposes, as indicated below. They should be considered subject to change in future releases, in either functionality or calling sequence, and possibly subject to removal if conditions warrant. In addition, advanced routines typically demand an understanding of the algorithms used by CPLEX, and thus incur a heightened risk of incorrect behavior in your application program—behavior that can be difficult to debug. Therefore, we encourage users to carefully consider whether they can accomplish the same task using the standard CPLEX API before using these advanced routines.

Designating and Modifying User-Written Callbacks in MIPs

- `CPXsetbranchcallbackfunc`
- `CPXsetcutcallbackfunc`
- `CPXsetdeletenodecallbackfunc`
- `CPXsetheuristiccallbackfunc`
- `CPXsetincumbentcallbackfunc`
- `CPXsetnodecallbackfunc`
- `CPXsetsolvecallbackfunc`

Accessing Current User-Written Callbacks in MIPs

- `CPXgetbranchcallbackfunc`
- `CPXgetcutcallbackfunc`
- `CPXgetdeletenodecallbackfunc`
- `CPXgetheuristiccallbackfunc`
- `CPXgetincumbentcallbackfunc`
- `CPXgetnodecallbackfunc`
- `CPXgetsolvecallbackfunc`

Specifying Actions with User-Written Callbacks with Routines

- `CPXbranchcallbackbranchbds`
- `CPXbranchcallbackbranchconstraints`
- `CPXbranchcallbackbranchgeneral`
- `CPXcutcallbackadd`
- `CPXcutcallbackaddlocal`

Querying from Within User-Written Callbacks in MIPs

- `CPXgetcallbacktype`
- `CPXgetcallbackgloballb`
- `CPXgetcallbackglobalub`

- CPXgetcallbackincumbent
- CPXgetcallbacklp
- CPXgetcallbacknodeinfo
- CPXgetcallbacknodeintfeas
- CPXgetcallbacknodelb
- CPXgetcallbacknodelp
- CPXgetcallbacknodeobjval
- CPXgetcallbacknodeub
- CPXgetcallbacknodex
- CPXgetcallbackorder
- CPXgetcallbackpseudocosts
- CPXgetcallbackseqinfo
- CPXgetcallbacksosinfo

Detecting Round-Off Errors in a Solution

- CPXcheckax
- CPXcheckpib

Using the Basis

- CPXbinvacol
- CPXbinvarow
- CPXbinvcol
- CPXbinvrow
- CPXbtran
- CPXftran
- CPXgetbhead
- CPXgetExactkappa
- CPXgetijrow
- CPXgetkappa

Accessing and Manipulating Norms

- CPXgetbasednorms
- CPXcopybasednorms
- CPXgetdnorms
- CPXcopydnorms
- CPXkilldnorms
- CPXgetpnorms
- CPXcopypnorms
- CPXkillpnorms

Accessing Information About the Current Problem

- CPXgetijdiv
- CPXgetobjoffset

Changing the Basis by Pivoting

- CPXpivot
- CPXpivotin
- CPXpivotout

Using User Cuts and Lazy Constraints

- CPXaddusercuts
- CPXfreeusercuts
- CPXaddlazyconstraints
- CPXfreelazyconstraints

Advanced Presolve Routines

- CPXbasicpresolve

- `CPXcopyprotected`
- `CPXcrushform`
- `CPXcrushpi`
- `CPXcrushx`
- `CPXfreepresolve`
- `CPXgetprestat`
- `CPXgetprotected`
- `CPXgetredlp`
- `CPXpreaddrows`
- `CPXprechgobj`
- `CPXpresolve`
- `CPXqpuncrushpi`
- `CPXuncrushform`
- `CPXuncrushpi`
- `CPXuncrushx`

Miscellaneous Routines

- `CPXcopypartialbase`
- `CPXdjfrompi`
- `CPXdualfarkas`
- `CPXgetray`
- `CPXmdleave`
- `CPXqpdjfrompi`
- `CPXslackfromx`
- `CPXsolwrite`
- `CPXstrongbranch`
- `CPXtightenbds`
- `CPXunscaleprob`

Concert Technology Library

- `IloCplex`, adding the advanced reference methods:

- `addLazyConstraint`
- `addLazyConstraints`
- `addUserCut`
- `addUserCuts`
- `basicPresolve`
- `clearLazyConstraints`
- `clearUserCuts`
- `importModel` with additional parameters

◆ `IloCplex::LazyConstraintCallbackI`

◆ `IloCplex::UserCutCallbackI`

◆ `ILOLAZYCONSTRAINTCALLBACK`

◆ `ILOUSERCUTCALLBACK`

Part I

Concepts

Advanced Presolve Functions

This chapter describes how to use the advanced presolve functions. The topics are:

- Introduction to Presolve
- Restricting Presolve Reductions
- Manual Control of Presolve
- Modifying a Problem

Introduction to Presolve

We begin our discussion of the advanced presolve interface with a quick introduction to presolve. Most of the information in this section will be familiar to people who are interested in the advanced interface, but we encourage everyone to read through it nonetheless.

As most CPLEX users know, presolve is a process whereby the problem input by the user is examined for logical reduction opportunities. The goal is to reduce the size of the problem passed to the requested optimizer. A reduction in problem size typically translates to a reduction in total run time (even including the time spent in presolve itself).

Consider `scorpion.mps` from NETLIB as an example:

```
CPLEX> disp pr st
Problem name: scorpion.mps
Constraints        :     388  [Less: 48,  Greater: 60,  Equal: 280]
Variables          :     358
Constraint nonzeros:    1426
Objective  nonzeros:     282
RHS        nonzeros:      76
CPLEX> optimize
Tried aggregator 1 time.
LP Presolve eliminated 138 rows and 82 columns.
Aggregator did 193 substitutions.
Reduced LP has 57 rows, 83 columns, and 327 nonzeros.
Presolve time =    0.00 sec.

Iteration log . . .
Iteration:     1   Dual objective     =           317.965093

Dual - Optimal:  Objective =    1.8781248227e+03
Solution time =    0.01 sec.  Iterations = 54 (0)
```

CPLEX is presented with a problem with 388 constraints and 358 variables, and after presolve the dual simplex method is presented with a problem with 57 constraints and 83 variables. Dual simplex solves this problem and passes the solution back to the presolve routine, which then unpresolves the solution to produce a solution to the original problem. During this process, presolve builds an entirely new 'presolved' problem and stores enough information to translate a solution to this problem back to a solution to the original problem. This information is hidden within the user's problem (in the CPLEX LP problem object, for Callable Library users) and was inaccessible to the user in CPLEX releases prior to 7.0.

The presolve process for a mixed integer program is similar, but has a few important differences. First, the actual presolve reductions differ. Integrality restrictions allow CPLEX to perform several classes of reductions that are invalid for continuous variables. A second difference is that the MIP solution process involves a series of linear program solutions. In the MIP branch & cut tree, a linear program is solved at each node. MIP presolve is performed once, at the beginning of the optimization (unless the `CPX_PARAM_RELAXPREIND` parameter is set, in which case the root relaxation is presolved a second time), and all of the node relaxation solutions use the presolved problem. Again, presolve stores the presolved problem and the information required to convert a presolved solution to a solution for the original problem within the LP problem object. Again, this information was inaccessible to the user in CPLEX releases prior to version 7.0.

A Proposed Example

Now consider an application where the user wishes to solve a linear program using the simplex method, then modify the problem slightly and solve the modified problem. As an example, let's say a user wishes to add a few new constraints to a problem based on the results of the first solution. The second solution should ideally start from the basis of the first, since starting from an advanced basis is usually significantly faster if the problem is only modified slightly.

Unfortunately, this scenario presents several difficulties. First, presolve must translate the new constraints on the original problem into constraints on the presolved problem. Presolve in releases prior to 7.0 could not do this. In addition, the new constraints may invalidate earlier presolve reductions, thus rendering the presolved problem useless for the reoptimization (an example is shown in "Restricting Presolve Reductions"). Presolve in releases prior to 7.0 had no way of disabling such reductions. In the prior releases, a user could either restart the optimization on the original, unpresolved problem or perform a new presolve on the modified problem. In the former case, the reoptimization does not get the problem size reduction benefits of presolve. In the latter, the second optimization does not obtain the benefit of having an advanced starting solution.

The advanced presolve interface can potentially make this and many other sequences of operations more efficient. It provides facilities to restrict the set of presolve reductions performed so that subsequent problem modifications can be accommodated. It also provides routines to translate constraints on the original problem to constraints on the presolved problem, so new constraints can be added to the presolved problem. As we discuss in "Additional Sections," it provides a variety of other capabilities.

When considering mixed integer programs, the advanced presolve interface plays a very different role. The branch & cut process needs to be restarted from scratch when the problem is even slightly modified, so preserving advanced start information isn't useful. The main benefit of the advanced presolve interface for MIPs is that it allows a user to translate decisions made during the branch & cut process (and based on the presolved problem) back to the corresponding constraints and variables in the original problem. This makes it easier for a user to control the branch & cut process. Details on the advanced MIP callback interface are provided in Chapter 2, *Advanced MIP Control Interface*.

Additional Sections

The organization of the remainder of this document is as follows. "Restricting Presolve Reductions" discusses the need for restricting the reductions performed by presolve, and the mechanisms the advanced presolve interface provides for doing so. "Manual Control of Presolve" discusses routines for manually controlling presolve. "Modifying a Problem" discusses routines for modifying a problem while still retaining presolve information.

Part II, *Routines*, includes detailed descriptions of each of these routines. Note that this current chapter discusses the routines without presenting them in full detail. Readers should look to Part II for details on the routines.

Restricting Presolve Reductions

As mentioned in the Introduction to Presolve, some presolve reductions are invalidated when a problem is modified. The advanced presolve interface therefore allows a user to tell presolve what sort of modifications will be made in the future, so presolve can avoid possibly invalid reductions. These considerations only apply to linear programs. The

presolved problem for quadratic programs cannot be modified because of the more complex presolve that is applied to quadratic programs.

Example: Adding Constraints to the First Solution

Let us reconsider our proposed example of adding a constraint to a problem after solving it. Imagine that you wish to optimize a linear program:

Primal:

$$\begin{array}{lrcrcrcr}
\textbf{max} & -x1 & + & x2 & + & x3 & & \\
\textbf{st} & x1 & + & x2 & + & 2x3 & \le & 6 \\
 & & & x2 & + & x3 & \le & 5 \\
 & & & & & & & 0 \\
 & x1, & & x2, & & x3 & \ge & 0
\end{array}$$

Dual:

$$\begin{array}{lrcrcrcr}
\textbf{min} & 6y1 & + & 5y2 & & & & \\
\textbf{st} & y1 & & & & & \ge & -1 \\
 & y1 & + & y2 & & & \ge & 1 \\
 & 2y1 & + & y2 & & & \ge & 1 \\
 & y1, & & y2, & & y3 & \ge & 0
\end{array}$$

Note that the first constraint in the dual ($y1 \ge -1$) is redundant. Presolve can use this information about the dual problem (and complementary slackness) to conclude that variable *x1* can be fixed to 0 and removed from the presolved problem. While it may be intuitively obvious from inspection of the primal problem that *x1* can be fixed to 0, it is important to note that dual information (redundancy of the first dual constraint) is used to formally prove it.

Now consider the addition of a new constraint $x2 \le 5x1$:

Primal:

$$\begin{array}{lrcrcrcr}
\textbf{max} & -x1 & + & x2 & + & x3 & & \\
\textbf{st} & x1 & + & x2 & + & 2x3 & \le & 6 \\
 & & & x2 & + & x3 & \le & 5 \\
 & -5x1 & + & x2 & & & \le & 0 \\
 & x1, & & x2, & & x3 & \ge & 0
\end{array}$$

Dual:

$$\begin{array}{lrcrcrcr}
\textbf{min} & 6y1 & + & 5y2 & & & & \\
\textbf{st} & y1 & & & - & 5y3 & \ge & -1 \\
 & y1 & + & y2 & + & y3 & \ge & 1 \\
 & 2y1 & + & y2 & & & \ge & 1 \\
 & y1, & & y2, & & y3 & \ge & 0
\end{array}$$

Our goal is to add the appropriate constraint to the presolved problem and reoptimize. Note, however, that the dual information presolve used to fix *x1* to 0 was changed by the addition of the new constraint. The first constraint in the dual is no longer guaranteed to be redundant, so the original fixing is no longer valid (the optimal solution is now $x1=1$, $x2=5$, $x3=0$). As a result, CPLEX is unable to use the presolved problem to reoptimize.

We classify presolve reductions into several classes: those that rely on primal information, those that rely on dual information, and those that rely on both. Future addition of new constraints, modifications to objective coefficients, and tightening of variable bounds (a special class of adding new constraints) require the user to turn off dual reductions. Introduction of new columns, modifications to right-hand-side values, and relaxation of

variable bounds (a special case of modifying right-hand-side values) require the user to turn off primal reductions.

These reductions are controlled through the CPX_PARAM_REDUCE parameter. The parameter has four possible settings. The default value CPX_PREREDUCE_PRIMALANDDUAL (3) indicates that presolve can rely on primal and dual information. With setting CPX_PREREDUCE_DUALONLY (2), presolve only uses dual information, with setting CPX_PREREDUCE_PRIMALONLY (1) it only uses primal information, and with setting CPX_PREREDUCE_NO_PRIMALORDUAL (0) it uses neither (which is equivalent to turning presolve off).

Setting the CPX_PARAM_REDUCE parameter has one additional effect on the optimization. Normally, the presolved problem and the presolved solution are freed at the end of an optimization call. However, when CPX_PARAM_REDUCE is set to a value other than its default, CPLEX assumes that the problem will subsequently be modified and reoptimized. It therefore retains the presolved problem and any presolved solution information (internally to the LP problem object). If the user has set CPX_PARAM_REDUCE and is finished with problem modification, he can call CPXfreepresolve() to free the presolved problem and reclaim the associated memory. The presolved problem is automatically freed when the user calls CPXfreeprob() on the original problem.

We should note that cutting planes in mixed integer programming are handled somewhat differently than one might expect. If a user wishes to add his own cuts during the branch & cut process (through CPXaddusercuts() or CPXcutcallbackadd()), it may appear necessary to turn off dual reductions to accommodate them. However, for reasons that are complex and beyond the scope of this discussion, dual reductions can be left on. The reasons relate to the fact that valid cuts never exclude integer feasible solutions, so dual reductions performed for the original problem are still valid after cutting planes are applied. However, a small set of reductions does need to be turned off. Recall that presolve must translate a new constraint on the original problem into a constraint on variables in the presolved problem. Most reductions performed by CPLEX presolve replace variables with linear expressions of zero or more other variables (plus a constant). A few do not. These latter reductions make it impossible to perform the translation to the presolved problem. Set CPX_PARAM_PRELINEAR to 0 to forbid these latter reductions.

Restricting the type of presolve reductions will also allow presolve to conclude more about infeasible and/or unbounded problems. Under the default setting of CPX_PARAM_REDUCE, presolve can only conclude that a problem is infeasible and/or unbounded. If CPX_PARAM_REDUCE is set to CPX_PREREDUCE_PRIMALONLY (1), presolve can conclude that a problem is primal infeasible with return status CPXERR_PRESLV_INF. If CPX_PARAM_REDUCE is set to CPX_PREREDUCE_DUALONLY (2), presolve can conclude that a problem is primal unbounded (if it is primal feasible) with return status CPXERR_PRESLV_UNBD.

A final facility that modifies the set of reductions performed by presolve is the CPXcopyprotected() routine. The user provides as input a list of variables in the original

problem that should survive presolve (that is, should exist in the presolved problem). Presolve will avoid reductions that remove those variables, with one exception. If a protected variable can be fixed, presolve will still remove it from the problem. This command is useful in cases where the user wants to explicitly control some aspect of the branch & cut process (for example, through the branch callback) using knowledge about those variables.

Manual Control of Presolve

While presolve was a highly automated and transparent procedure in releases of CPLEX prior to 7.0, releases 7.0 and above give the user significant control over when presolve is performed and what is done with the result. This section discusses these added control facilities. Recall that the functions mentioned here are described in detail, including arguments and return values, in Part II, *Routines*.

The first control function provided by the advanced presolve interface is `CPXpresolve()`, which manually invokes presolve on the supplied problem. Once this routine is called on a problem, the problem has a presolved problem associated with it. Subsequent calls to optimization routines (`CPXprimopt()`, `CPXdualopt()`, `CPXbaropt()`, `CPXmipopt()`) will use this presolved problem without repeating the presolve, provided no operation that discards the presolved problem is performed in the interim. The presolved problem is automatically discarded if a problem modification is performed that is incompatible with the setting of `CPX_PARAM_REDUCE` (further information is provided in "Modifying a Problem").

By using the `CPX_PARAM_REDUCE` to restrict the types of presolve reductions, CPLEX can make use of the optimal basis to the presolved problem. If you set `CPX_PARAM_REDUCE` to restrict presolve reductions, then make problem modifications that don't invalidate those reductions, CPLEX will automatically use the optimal basis to the presolved problem. On the other hand, if the nature of the problem modifications is such that you cannot set `CPX_PARAM_REDUCE`, you can still perform an advanced start by making the modifications, calling `CPXpresolve()` to create the new presolved problem, then calling `CPXcopystart()`, passing the original model and any combination of primal and dual solutions. CPLEX will crush the solutions and use them to construct a starting basis for the presolved model.

We should point out a few of the subtleties associated with using `CPXcopystart()` to start an optimization from an advanced, presolved solution. This routine only creates a presolved solution vector if the presolved problem is already present (either because the user called `CPXpresolve()` or because the user turned off some presolve reductions through `CPX_PARAM_REDUCE` and then solved a problem). The earlier sequence would not have started from an advanced solution if `CPXcopystart()` had been called before `CPXpresolve()`. Another important detail about `CPXcopystart()` is that it crushes primal and/or dual solutions, not bases. It then uses the crushed solutions to choose a starting basis. If you have a basis, you need to compute optimal primal and dual solutions (using

`CPXcopybase()` and then `CPXprimopt()`), extract them, and then call `CPXcopystart()` with them to use the corresponding advanced solution. In general, starting with both a primal and dual solution is preferable to starting with one or the other. One other thing to note about `CPXcopystart()` is that the primal and dual slack (slack and dj) arguments are optional. The routine will compute slacks values if none are provided.

Another situation where a user might want to use `CPXpresolve()` is if the user wishes to gather information about the presolve, possibly in preparation for using the advanced MIP callback routines to control the branch & cut process. Once `CPXpresolve()` has been called, the user can then call `CPXgetprestat()` to obtain information about the reductions performed on the problem. This function provides information, for each variable in the original problem, on whether the variable was fixed and removed, aggregated out, removed for some other reason, or is still present in the reduced problem. It also gives information, for each row in the original problem, on whether it was removed due to redundancy, aggregated out, or is still present in the reduced problem. For those rows and columns that are present in the reduced problem, this function provides a mapping from original row/column number to row/column number in the reduced problem, and vice-versa.

Another situation where a user might want to use `CPXpresolve()` is to work directly on the presolved problem. By calling `CPXgetredlp()` immediately after `CPXpresolve()`, the user can obtain a pointer to the presolved problem. For an example of how this might be used, the user could call routines `CPXcrushx()` and `CPXcrushpi()` to presolve primal and dual solution vectors, call `CPXgetredlp()` to get access to the presolved problem, then use `CPXcopystart()` to copy the presolved solutions into the presolved problem, then optimize the problem, and finally call routines `CPXuncrushx()` and `CPXuncrushpi()`—`CPXqpuncruspi()` for QPs—to unpresolve solutions from the presolved problem, creating solutions for the original problem.

Please note that `CPXgetredlp()` provides the user access to internal CPLEX data structures. The presolved problem MUST NOT be modified by the user. If the user wishes to manipulate the reduced problem, the user should make a copy of it (using `CPXcloneprob()`) and manipulate the copy instead.

The advanced presolve interface provides another call that is useful when working directly with the presolved problem (either through `CPXgetredlp()` or through the advanced MIP callbacks). The `CPXcrushform()` call translates a linear expression in the original problem into a linear expression in the presolved problem. The most likely use of this routine is to add user cuts to the presolved problem during a mixed integer optimization. The advanced presolve interface also provides the reverse operation. The `CPXuncrushform()` routine translates a linear expression in the presolved problem into a linear expression in the original problem.

A limited presolve analysis is performed by `CPXbasicpresolve()` and `IloCplex::basicPresolve`. This function determines which rows are redundant and computes strengthened bounds on the variables. This information can be used to derive some types of cuts that will tighten the formulation, to aid in formulation by pointing out

redundancies, and to provide upper bounds for variables that must have them, like semi-continuous variables. `CPXbasicpresolve()` does not create a presolved problem.

The interface allows the user to manually free the memory associated with the presolved problem using routine `CPXfreepresolve()`. The next optimization call (or call to `CPXpresolve()`) recreates the presolved problem.

Modifying a Problem

This section briefly discusses the mechanics of modifying a problem after presolve has been performed. This discussion applies only to linear programs and mixed integer programs; it does not apply to quadratic programs.

As noted earlier, the user must indicate through the `CPX_PARAM_REDUCE` parameter the types of modifications that are going to be performed to the problem. Recall that if primal reductions are turned off, the user can add variables, change the right-hand-side vector, or loosen variable bounds without losing the presolved problem. These changes are made through the standard problem modification interface (`CPXaddcols()`, `CPXchgrhs()`, and `CPXchgbds()`).

Recall that if dual reductions are turned off, the user can add constraints to the problem, change the objective function, or tighten variable bounds. Variable bounds are tightened through the standard interface (`CPXchgbds()`). The addition of constraints or changes to the objective value must be done through the two interface routines `CPXpreaddrows()` and `CPXprechgobj()`. We should note that the constraints added by `CPXpreaddrows()` are equivalent to but sometimes different from those input by the user. The dual variables associated with the added rows may take different values than those the user might expect.

If a user makes a problem modification that is not consistent with the setting of `CPX_PARAM_REDUCE`, the presolved problem is discarded and presolve is reinvoked at the next optimization call. Similarly, CPLEX discards the presolved problem if the user modifies a variable or constraint that presolve had previously removed from the problem. You can use `CPXpreaddrows()` or `CPXprechgobj()` to make sure that this will not happen. Note that `CPXpreaddrows()` also permits changes to the bounds of the presolved problem. If the nature of the procedure dictates a real need to modify the variables that presolve removed, you can use the `CPXcopyprotected()` routine to instruct CPLEX not to remove those variables from the problem.

Instead of changing the bounds on the presolved problem, consider changing the bounds on the original problem. CPLEX will discard the presolved problem, but calling `CPXpresolve()` will cause CPLEX to apply presolve to the modified problem, with the added benefit of reductions based on the latest problem modifications. Then use `CPXcrushx()`, `CPXcrushpi()`, and `CPXcopystart()` to provide an advanced start for the problem after presolve has been applied on the modified problem.

2

Advanced MIP Control Interface

This chapter describes the CPLEX 8.1 advanced MIP control interface. It includes sections on:

- Introduction to MIP Callbacks
- Heuristic Callback
- Cut Callback
- Branch Selection Callback
- Incumbent Callback
- Node Selection Callback
- Solve Callback

These callbacks allow sophisticated users to control the details of the branch & cut process. Specifically, users can choose the next node to explore, choose the branching variable, add their own cutting planes, place additional restrictions on integer solutions, or insert their own heuristic solutions. These functions are meant for situations where other tactics to improve CPLEX's performance on a hard MIP problem, such as non-default parameter settings or priority orders, have failed. We refer the reader to the section on "Troubleshooting MIP Performance Problems" in the *ILOG CPLEX User's Manual* for more information on MIP parameters and priority orders.

Users of the advanced MIP control interface can work with the variables of the presolved problem or, by following a few simple rules, can instead work with the variables of the original problem.

> ***Tip:*** The advanced MIP control interface relies heavily on the advanced presolve capabilities. We suggest that the reader become familiar with Chapter 1, *Advanced Presolve Functions*, before reading this chapter.

Control callbacks in the ILOG Concert Technology CPLEX Library use original model variables. These callbacks are fully documented in the *ILOG CPLEX Reference Manual*, except for the callbacks `IloCplex::UserCutCallbackI` and `IloCplex::LazyConstraintCallbackI`, which are documented here.

Introduction to MIP Callbacks

As the reader is no doubt familiar, the process of solving a mixed integer programming problem involves exploring a tree of linear programming relaxations. CPLEX repeatedly selects a node from the tree, solves the LP relaxation at that node, attempts to generate cutting planes to cut off the current solution, invokes a heuristic to try to find an integer feasible solution "close" to the current relaxation solution, selects a branching variable (an integer variable whose value in the current relaxation is fractional), and finally places the two nodes that result from branching up or down on the branching variable back into the tree.

The CPLEX Mixed Integer Optimizer includes methods for each of the important steps listed above (node selection, cutting planes, heuristic, branch variable selection, incumbent replacement). While default CPLEX methods are generally effective, and parameters are available to choose alternatives if the defaults are not working for a particular problem, there are rare cases where a user may wish to influence or even override CPLEX methods. CPLEX provides a callback mechanism to allow the user to do this. If the user installs a callback routine, CPLEX calls this routine during the branch & cut process to allow the user to intervene. CPLEX callback functions are thread-safe for use in parallel (multiple CPU) applications.

Before describing the callback routines, we first discuss an important issue related to presolve that the user should be aware of. Most of the decisions made within MIP relate to the variables of the problem. The heuristic, for example, finds values for all the variables in the problem that produce a feasible solution. Similarly, branching chooses a variable on which to branch. When considering user callbacks, the difficulty that arises is that the user is familiar with the variables in the original problem, while the branch & cut process is performed on the presolved problem. Many of the variables in the original problem may have been modified or removed by presolve.

CPLEX provides two options for handling the problem of mapping from the original problem to the presolved problem. First, the user may work directly with the presolved problem and presolved solution vectors. This is the default. While this option may at first appear unwieldy, note that the Advanced Presolve Interface allows the user to map between original variables and presolved variables. The downside to this option is that the user has to manually invoke these advanced presolve routines. The second option is to set `CPX_PARAM_MIPCBREDLP` to `CPX_OFF` (0), thus requesting that the callback routines work exclusively with original variables. CPLEX automatically translates the data between original and presolved data. While the second option is simpler, the first provides more control. These two options will be revisited at several points in this chapter.

Heuristic Callback

The first user callback we consider is the heuristic callback. The first step in using this callback is to call `CPXsetheuristiccallbackfunc()`, with a pointer to a callback function and optionally a pointer to user private data as arguments. We refer the reader to advanced example `admipex2.c` for further details of how this callback is used. Once this routine has been called, CPLEX calls the user callback function at every viable node in the branch & cut tree (we call a node viable if its LP relaxation is feasible and its relaxation objective value is better than that of the best available integer solution). The user callback routine is called with the solution vector for the current relaxation as input. The callback function should return a feasible solution vector, if one is found, as output.

The advanced MIP control interface provides several routines that allow the user callback to gather information that may be useful in finding heuristic solutions. The routines `CPXgetcallbackgloballb()` and `CPXgetcallbackglobalub()`, for example, return the tightest known global lower and upper bounds on all the variables in the problem. No feasible solution whose objective is better than that of the best known solution can violate these bounds. Similarly, the routines `CPXgetcallbacknodelb()` and `CPXgetcallbacknodeub()` return variable bounds at this node. These reflect the bound adjustments made during branching. The routine `CPXgetcallbackincumbent()` returns the current incumbent - the best known feasible solution. The routine `CPXgetcallbacklp()` returns a pointer to the MIP problem (presolved or unpresolved, depending on the `CPX_PARAM_MIPCBREDLP` parameter). This pointer can be used to obtain various information about the problem (variable types, etc.), or as an argument for the advanced presolve interface if the user wishes to manually translate between presolved and unpresolved values. In addition, the callback can use the `cbdata` parameter passed to it, along with routine `CPXgetcallbacknodelp()`, to obtain a pointer to the node relaxation LP. This can be used to access desired information about the relaxation (row count, column count, etc.). Note that in both cases, the user should never use the pointers obtained from these callbacks to modify the associated problems.

As noted earlier, the CPX_PARAM_MIPCBREDLP parameter influences the arguments to the user callback routine. If this parameter is set to its default value of CPX_ON (1), the solution vector returned to the callback, and any feasible solutions returned by the callback, are presolved vectors. They contain one value for each variable in the presolved problem. The same is true of the various callback support routines (CPXgetcallbackgloballb(), etc.). If the parameter is set to CPX_OFF (0), all these vectors relate to variables of the original problem. Note that this parameter should not be changed in the middle of an optimization.

The user should be aware that the branch & cut process works with the presolved problem, so the code will incur some cost when translating from presolved to original values. This cost is usually small, but can sometimes be significant.

We should also note that if a user wishes to solve linear programs as part of a heuristic callback, the user must make a copy of the node LP (for example, using CPXcloneprob()). The user should not modify the CPLEX node LP.

Cut Callback

The next example we consider is the user cut callback routine. The user calls CPXsetcutcallbackfunc() to set a cut callback, and the user's callback routine is called at every viable node of the branch & cut tree. We refer the reader to admipex5.c for a detailed example.

A likely sequence of events once the user callback function is called is as follows. First, the routine calls CPXgetcallbacknodex() to get the relaxation solution for the current node. It possibly also gathers other information about the problem (through CPXgetcallbacklp(), CPXgetcallbackgloballb(), etc.) It then calls a user separation routine to identify violated user cuts. These cuts are then added to the problem by calling CPXcutcallbackadd(), and the callback returns. Local cuts, that is, cuts that apply to the subtree of which the current node is the root, can be added by calling CPXcutcallbackaddlocal().

At this point, it is important to draw a distinction between the two different types of constraints that can be added through the cut callback interface. The first type is the traditional MIP cutting plane, which is a constraint that can be derived from other constraints in the problem and does not cut off any integer feasible solutions. The second is a “lazy constraint”, which is a constraint that can not be derived from other constraints and potentially cuts off integer feasible solutions. Either type of constraint can be added through the cut callback interface.

As with the heuristic callback, the user can choose whether to work with presolved values or original values. If the user chooses to work with original values, a few parameters must be modified. If the user adds only cutting planes on the original problem, the user must set advanced presolve parameter CPX_PARAM_PRELINEAR to CPX_OFF (0). This parameter

forbids certain presolve reductions that make translation from original values to presolved values impossible.

If the user adds any lazy constraints, the user must turn off dual presolve reductions (using the `CPX_PARAM_REDUCE` parameter). The user must think carefully about whether constraints added through the cut interface are implied by existing constraints, in which case dual presolve reductions may be left on, or whether they are not, in which case dual reductions are forbidden.

ILOG Concert Technology users should use the `IloCplex::LazyConstraintCallbackI` when adding lazy constraints, and the `IloCplex::UserCutCallbackI` when adding cutting planes. Dual reductions and/or non-linear reductions then will be turned off automatically.

One scenario that merits special attention is when the user knows a large set of cuts a priori. Rather than adding them to the original problem, the user may instead wish to add them only when violated. The CPLEX advanced MIP control interface provides more than one mechanism for accomplishing this. The first and probably most obvious at this point is to install a user callback that checks each cut from the user set at each node, adding those that are violated. The user can do this either by setting `CPX_PARAM_MIPCBREDLP` to `CPX_OFF` to work with the original problem in the cut callback, or by using the Advanced Presolve Interface to translate the cuts on the original problem to cuts on the presolved problem, and then use the presolved cuts in the cut callback.

Another, perhaps simpler alternative is to add the cuts or constraints to cut pools before optimization begins. This is discussed in Chapter 3, *User Cut and Lazy Constraint Pools*.

Branch Selection Callback

The next callback we consider is the branch variable selection callback. After calling `CPXsetbranchcallbackfunc()` with a pointer to a user callback routine, the user routine is called whenever CPLEX makes a branching decision. CPLEX indicates which variable has been chosen for branching and allows the user to modify that decision. The user may specify the number of children for the current node (between 0 and 2), and the set of bounds and/or constraints that are modified for each child through calling `CPXbranchcallbackbranchbds()`, `CPXbranchcallbackbranchconstraints()`, or `CPXbranchcallbackbranchgeneral()`. The branch callback routine is called for all viable nodes. In particular, it will be called for nodes that have zero integer infeasibilities; in this case, CPLEX will not have chosen a branch variable, and the default action will be to discard the node. The user can choose to branch from this node and in this way impose additional restrictions on integer solutions.

A user branch routine may, for example, call `CPXgetcallbacknodeintfeas()` to identify branching candidates, call `CPXgetcallbackpseudocosts()` to obtain pseudo-cost

information on these variables, call `CPXgetcallbackorder()` to get priority order information, make a decision based on this and perhaps other information, and then respond that the current node will have two children, where one has a new lower bound on the branch variable and the other has a new upper bound on that variable.

Alternatively, the branch callback routine can be used to sculpt the search tree by pruning nodes or adjusting variable bounds. Choosing zero children for a node prunes that node, while choosing one node with a set of new variable bounds adjusts bounds on those variables for the entire subtree rooted at this node. Note that the user must be careful when using this routine for anything other than choosing a different variable to branch on. Pruning a valid node or placing an invalid bound on a variable can prune the optimal solution.

We should point out one important detail associated with the use of the `CPX_PARAM_MIPCBREDLP` parameter in a branch callback. If this parameter is set to `CPX_OFF` (0), the user can choose branch variables (and add bounds) for the original problem. However, not every fractional variable is actually available for branching. Recall that some variables are replaced by linear combinations of other variables in the presolved problem. Since branching involves adding new bounds to specific variables in the presolved problem, a variable must be present in the presolved problem for it to be branched on. The user should use the `CPXgetcallbacknodeintfeas()` routine from the Advanced Presolve Interface to find branching candidates (those for which `CPXgetcallbacknodeintfeas()` returns `CPX_INTEGER_INFEASIBLE`). The `CPXcopyprotected()` routine can be used to prevent presolve from removing specific variables from the presolved problem. While restricting branching may appear to limit your ability to solve a problem, in fact a problem can always be solved to optimality by branching only on the variables of the presolved problem.

Incumbent Callback

The incumbent callback is used to reject integer feasible solutions that do not meet additional restrictions the user may wish to impose. The user callback routine will be called each time a new incumbent solution has been found, including when solutions are provided by the user's heuristic callback routine. The user callback routine is called with the new solution as input. The callback function should return a value that indicates whether or not the new solution should replace the incumbent solution.

As with other MIP control callback routines, the `CPX_PARAM_MIPCBREDLP` parameter influences the arguments to the user callback routine. If this parameter is set to its default value of `CPX_ON (1)`, the solution vector that is input to the callback is a presolved vector. It contains one value for each variable in the presolved problem. The same is true of the various callback support routines (`CPXcallbackglobalub()`, and so forth.). If the parameter is set to `CPX_OFF (0)`, all these vectors relate to the variables of the original problem. Note that this parameter should not be changed in the middle of an optimization.

Node Selection Callback

The user can influence the order in which nodes are explored by installing a node selection callback (through `CPXsetnodecallbackfunc()`). When CPLEX chooses the node to explore next, it will call the user callback routine, with CPLEX's choice as an argument. The callback has the option of modifying this choice.

Solve Callback

The final callback we consider is the solve callback. By calling `CPXsetsolvecallbackfunc()`, the user instructs CPLEX to call a user function rather than the CPLEX choice (dual simplex by default) to solve the linear programming relaxations at each node of the tree. Advanced example `admipex6.c` gives an example of how this callback might be used.

> ***Note:*** *We expect the most common use of this callback will be to craft a customer solution strategy out of the set of available CPLEX algorithms. For example, a user might create a hybrid strategy that checks for network status, calling* `CPXhybnetopt()` *instead of* `CPXdualopt()` *when it finds it.*

3

User Cut and Lazy Constraint Pools

Sometimes a user may know a large set of cutting planes a priori (user cuts), or have additional constraints that are unlikely to be violated (lazy constraints). Simply including these cuts or constraints in the original formulation would make the LP subproblem of a MIP optimization very large and/or expensive to solve. Instead, these can be handled through the cut callback described in Chapter 2, *Advanced MIP Control Interface*, or by setting up cut pools before MIP optimization begins.

This chapter contains the sections:

- Adding User Cuts and Lazy Constraints
- Deleting User Cuts and Lazy Constraints

Adding User Cuts and Lazy Constraints

You may add user cuts or lazy constraints through `add` routines in the Component Libraries or via LP and SAV files.

Component Libraries

The following routines will add to the user cut pool.

- The CPLEX Callable Library routine is `CPXaddusercuts()`. CPLEX will scan the cut pool for violated cuts and add these to the LP subproblem, and no integer solution will violate the cuts.
- The Concert Technology routine is `IloCplex::addUserCuts()`.

The following routines will add to the lazy constraint pool.

- The CPLEX Callable Library routine is `CPXaddlazyconstraints()`. CPLEX will scan the pool for violated constraints and add these to the MIP subproblems, and no integer solution will violate the constraints. Additionally, the tight lazy constraints will be added to the fixed problem, obtained by calling the routine `CPXchgprobtype()`.
- The Concert Technology routine is `IloCplex::addLazyConstraints()`.

Note that CPLEX does not guarantee that user cuts and lazy constraints are added as soon as they are violated by a node relaxation. It simply guarantees that no feasible solutions returned by CPLEX will violate these constraints.

Reading LP and SAV Files

User cuts and lazy constraints may also be specified in LP-format and SAV-format files, and so may be read:

- With the Interactive Optimizer.
- Through the routines `CPXreadcopyprob()` and `IloCplex::importModel()`.

General Syntax

The general syntax rules for LP format given in the *ILOG CPLEX Reference Manual* apply to user cuts and lazy constraints.

- The user cuts section or sections must be preceded by the keywords `USER CUTS`.
- The lazy constraints section or sections must be preceded by the keywords `LAZY CONSTRAINTS`.

These sections, and the ordinary constraints section preceded by the keywords `SUBJECT TO`, can appear in any order and can be present multiple times, as long as they are placed after the objective function section and before any of the keywords `BOUNDS`, `GENERALS`, `BINARIES`, `SEMI-CONTINUOUS` or `END`.

Example

Here is an example of a file containing ordinary constraints and lazy constraints.

```
Maximize
```

```
 obj: 12 x1 + 5 x2 + 15 x3 + 10 x4
Subject To
 c1: 5 x1 + x2 + 9 x3 + 12 x4 <= 15
Lazy Constraints
 l1: 2 x1 + 3 x2 + 4 x3 + x4 <= 10
 l2: 3 x1 + 2 x2 + 4 x3 + 10 x4 <= 8
Bounds
 0 <= x1 <= 5
 0 <= x2 <= 5
 0 <= x3 <= 5
 0 <= x4 <= 5
Generals
 x1  x2  x3  x4
End
```

The optional constraint names in LP format for user cuts and lazy constraints are discarded.

Writing LP and SAV Files

When writing LP or SAV format files, user cuts and lazy constraints added through their respective `add` routines or read from LP or SAV format files will be included in the output files. In LP-format files:

- User cuts will be given names of the form `u`*x* where *x* is an index number starting at 1.
- Lazy constraints will be given names of `l`*x* where *x* is again an index number starting at 1.

Using the Interactive Optimizer

User cuts and lazy constraints will appear when the command `display problem all` is issued in the Interactive Optimizer. User cuts and lazy constraints can also be added to an existing problem with the `add` command of the Interactive Optimizer.

CPLEX Parameters

In the Callable Library, CPLEX parameters must be set as follows:

- When a user cut pool is present, the `CPX_PARAM_PRELINEAR` parameter must be set to 0.
- When a lazy constraint pool is present, the `CPX_PARAM_REDUCE` parameter must be set to `CPX_PREREDUCE_PRIMALONLY (1)` or `CPX_PREREDUCE_NOPRIMALORDUAL (0)`.

If these parameters do not have these values, the error `CPXERR_PRESOLVE_BAD_PARAM` will be issued when `CPXmipopt()` is called.

The Concert Technology CPLEX Library will automatically handle these parameter settings.

User Cut and Lazy Constraint Pools

Deleting User Cuts and Lazy Constraints

The user cut and lazy constraint pools are cleared by calling the routines `CPXfreeusercuts()` and `CPXfreelazyconstraints()`. Clearing the pools will not change the MIP solution.

The Concert Technology routines are `IloCplex::clearUserCuts()` and `IloCplex::clearLazyConstraints()`.

Part II

Routines

CPXaddlazyconstraints

Usage

Mixed Integer Users Only

Description

The routine `CPXaddlazyconstraints()` is used to add constraints to the list of constraints that should be added to the LP subproblem of a MIP optimization if they are violated. CPLEX handles addition of the constraints and makes sure that all integer solutions satisfy all the constraints. The constraints are added to those specified in prior calls to `CPXaddlazyconstraints()`.

Lazy constraints are constraints not specified in the constraint matrix of the MIP problem, but that must be not be violated in a solution. Using lazy constraints makes sense when there are a large number of constraints that must be satisfied at a solution, but are unlikely to be violated if they are left out.

The CPLEX parameter `CPX_PARAM_REDUCE` should be set to `CPX_PREREDUCE_NOPRIMALORDUAL (0)` or to `CPX_PREREDUCE_PRIMALONLY (1)` in order to turn off dual reductions.

Use `CPXfreelazyconstraints()` to clear the list of lazy constraints.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXaddlazyconstraints (CPXCENVptr env,
                           CPXLPptr lp,
                           int rcnt,
                           int nzcnt,
                           const double *rhs,
                           const char *sense,
                           const int *rmatbeg,
                           const int *rmatind,
                           const double *rmatval);
```

Arguments

The arguments of `CPXaddlazyconstraints()` are the same as those of `CPXaddrows()`, with the exception that new columns may not be specified, so there are no `ccnt` and `colname` arguments, and row names may not be specified, so there is no `rowname` argument.

Example

```
status = CPXaddlazyconstraints (env, lp, cnt, nzcnt, rhs, sense,
                                beg, ind, val);
```

CPXaddusercuts

Usage

Mixed Integer Users Only

Description

The routine `CPXaddusercuts()` is used to add constraints to the list of constraints that should be added to the LP subproblem of a MIP optimization if they are violated. CPLEX handles addition of the constraints and makes sure that all integer solutions satisfy all the constraints. The constraints are added to those specified in prior calls to `CPXaddusercuts()`.

The constraints must be cuts, which are implied by the constraint matrix. The CPLEX parameter `CPX_PARAM_PRELINEAR` should be set to `CPX_OFF` (0).

Use `CPXfreeusercuts()` to clear the list of cuts.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXaddusercuts (CPXCENVptr env,
                    CPXLPptr lp,
                    int rcnt,
                    int nzcnt,
                    const double *rhs,
                    const char *sense,
                    const int *rmatbeg,
                    const int *rmatind,
                    const double *rmatval);
```

Arguments

The arguments of `CPXaddusercuts()` are the same as those of `CPXaddrows()`, with the exception that new columns may not be specified, so there are no `ccnt` and `colname` arguments, and row names may not be specified, so there is no `rowname` argument.

Example

```
status = CPXaddusercuts (env, lp, cutcnt, cutnzcnt, cutrhs,
                         cutsense, cutbeg, cutind, cutval);
```

See Also

Example `admipex4.c` *in the advanced examples directory*

CPXbasicpresolve

Usage

Advanced

Description

The routine `CPXbasicpresolve()` performs bound strengthening and detects redundant rows. `CPXbasicpresolve()` does not create a presolved problem. This routine cannot be used for quadratic programs.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXbasicpresolve (CPXCENVptr env,
                      CPXLPptr lp,
                      double *redlb,
                      double *redub,
                      int *rstat);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`double *redlb`

An array to receive the strengthened lower bounds. The array must be of length at least the number of columns in the LP problem object. May be `NULL`.

`double *redub`

An array to receive the strengthened upper bounds. The array must be of length at least the number of columns in the LP problem object. May be `NULL`.

`int *rstat`

An array to receive the status of the row. The array must be of length at least the number of rows in the LP problem object. May be `NULL`.

Values for `rstat[i]`:

0 if row `i` is not redundant

-1 if row `i` is redundant

Example

```
status = CPXbasicpresolve (env, lp, reducelb, reduceub, rowstat);
```

CPXbinvacol

Usage

Advanced

Description

The routine `CPXbinvacol()` computes the representation of the j^{th} column in terms of the basis. In other words, it solves $\boldsymbol{B}x = \boldsymbol{A}j$.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXbinvacol (CPXCENVptr env,
                 CPXCLPptr lp,
                 int j,
                 double *x);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int j`

An integer that indicates the index of the column to be computed.

`double *x`

An array containing the solution of $\boldsymbol{B}x = \boldsymbol{A}j$. The array must be of length at least equal to the number of rows in the problem.

CPXbinvarow

Usage

Advanced

Description

The routine `CPXbinvarow()` computes the i^{th} row of $\boldsymbol{B}^{-1}\boldsymbol{A}$. In other words, it computes the i^{th} row of the tableau.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXbinvarow (CPXCENVptr env,
                 CPXCLPptr lp,
                 int i,
                 double *z);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int i`

An integer that indicates the index of the row to be computed.

`double *z`

An array containing the i^{th} row of $\boldsymbol{B}^{-1}\boldsymbol{A}$. The array must be of length at least equal to the number of columns in the problem.

CPXbinvcol

Usage

Advanced

Description

The routine `CPXbinvcol()` computes the j^{th} column of the basis inverse.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXbinvcol (CPXCENVptr env,
                CPXCLPptr lp,
                int j,
                double *x);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int j`

An integer that indicates the index of the column of the basis inverse to be computed.

`double *x`

An array containing the j^{th} column of $\boldsymbol{B}^{-1}$. The array must be of length at least equal to the number of rows in the problem.

CPXbinvrow

Usage Advanced

Description The routine `CPXbinvrow()` computes the i^{th} row of the basis inverse.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXbinvrow (CPXCENVptr env,
                CPXCLPptr lp,
                int i,
                double *y);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to the CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int i`

An integer that indicates the index of the row to be computed.

`double *y`

An array containing the i^{th} row of $\boldsymbol{B}^{-1}$. The array must be of length at least equal to the number of rows in the problem.

CPXbranchcallbackbranchbds

Usage

Mixed Integer Users Only

Description

The routine `CPXbranchcallbackbranchbds()`specifies the branches to be taken from the current node. It may be called only from within a user-written branch callback function.

Branch variables are in terms of the original problem if the parameter `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF` before the call to `CPXmipopt()` that calls the callback. Otherwise, branch variables are in terms of the presolved problem.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXbranchcallbackbranchbds (CPXCENVptr env,
                                void *cbdata,
                                int wherefrom,
                                double nodeest,
                                int cnt,
                                int *indices,
                                const char *lu,
                                const int *bd,
                                void *userhandle,
                                int *seqnum_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

A pointer passed to the user-written callback. This parameter must be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value that indicates where the user-written callback was called from. This parameter must be the value of `wherefrom` passed to the user-written callback.

`double nodeest`

A double that indicates the value of the node estimate for the node to be created with this branch. The node estimate is used to select nodes from the branch & cut tree with certain values of the `NodeSel` parameter.

`int cnt`

An integer that indicates the number of bound changes that are specified in the arrays `indices`, `lu`, and `bd`.

`int *indices`

Together with `lu` and `bd`, this array defines the bound changes for the branch. The entry `indices[i]` is the index for the variable.

`const char *lu`

Together with `indices` and `bd`, this array defines the bound changes for each of the created nodes. The entry `lu[i]` is one of the three possible values indicating which bound to change: `L` for lower bound, `U` for upper bound, or `B` for both bounds.

`const int *bd`

Together with `indices` and `lu`, this array defines the bound changes for each of the created nodes. The entry `bd[i]` indicates the new value of the bound.

`void *userhandle`

A pointer to user private data that should be associated with the node created by this branch. May be `NULL`.

`int *seqnum_p`

A pointer to an integer that, on return, will contain the sequence number that CPLEX has assigned to the node created from this branch. The sequence number may be used to select this node in later calls to the node callback.

CPXbranchcallbackbranchconstraints

Description

The routine `CPXbranchcallbackbranchconstraints()`specifies the branches to be taken from the current node when the branch is specified by adding one or more constraints to the node problem. It may be called only from within a user-written branch callback function.

Constraints are in terms of the original problem if the parameter `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF` before the call to `CPXmipopt()` that calls the callback. Otherwise, constraints are in terms of the presolved problem.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXbranchcallbackbranchconstraints (CPXCENVptr env,
                                        void *cbdata,
                                        int wherefrom,
                                        double nodeest,
                                        int rcnt,
                                        int nzcnt,
                                        const double *rhs,
                                        const char *sense,
                                        const int *rmatbeg,
                                        const int *rmatind,
                                        const double *rmatval,
                                        void *userhandle
                                        int *seqnum_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

A pointer passed to the user-written callback. This parameter must be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value that indicates where the user-written callback was called from. This parameter must be the value of `wherefrom` passed to the user-written callback.

`double nodeest`

A double that indicates the value of the node estimate for the node to be created with this branch. The node estimate is used to select nodes from the branch & cut tree with certain values of the `NodeSel` parameter.

`int rcnt`

An integer that indicates the number of constraints for the branch.

`int nzcnt`

An integer that indicates the number of nonzero constraint coefficients for the branch. This specifies the length of the arrays `rmatind` and `rmatval`.

`const double *rhs`

An array of length `rcnt` containing the right-hand side term for each constraint for the branch.

`const char *sense`

An array of length `rcnt` containing the sense of each constraint to be added for the branch.

`sense[i] = 'L'` $\leq$ `constraint`

`sense[i] = 'E' = constraint`

`sense[i] = 'G'` $\geq$ `constraint`

`const int *rmatbeg`

`const int *rmatind`

`const double *rmatval`

Arrays that describe the constraints for the branch. The format is similar to the format used to describe the constraint matrix in the routine `CPXaddrows`. Every row must be stored in sequential locations in this array from position `rmatbeg[i]` to `rmatbeg[i+1]-1` (or from `rmatbeg[i]` to `nzcnt -1` if `i=rcnt-1`).

Each entry, `rmatind[i]`, indicates the column index of the corresponding coefficient, `rmatval[i]`. All rows must be contiguous, and `rmatbeg[0]` must be `0`.

`void *userhandle`

A pointer to user private data that should be associated with the node created by this branch. May be `NULL`.

`int *seqnum_p`

A pointer to an integer that, on return, will contain the sequence number that CPLEX has assigned to the node created from this branch. The sequence number may be used to select this node in later calls to the node callback.

CPXbranchcallbackbranchgeneral

Description

The routine `CPXbranchcallbackbranchgeneral()`specifies the branches to be taken from the current node when the branch includes variable bound changes and additional constraints. It may be called only from within a user-written branch callback function.

Branch variables are in terms of the original problem if the parameter `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF` before the call to `CPXmipopt()` that calls the callback. Otherwise, branch variables are in terms of the presolved problem.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXbranchcallbackbranchgeneral (CPXCENVptr env,
                                    void *cbdata,
                                    int wherefrom,
                                    double nodeest,
                                    int varcnt,
                                    const int *varind,
                                    const char *varlu,
                                    const int *varbd,
                                    int rcnt,
                                    int nzcnt,
                                    const double *rhs,
                                    const char *sense,
                                    const int *rmatbeg,
                                    const int *rmatind,
                                    const double *rmatval,
                                    void *userhandle
                                    int *seqnum_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

A pointer passed to the user-written callback. This parameter must be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value that indicates where the user-written callback was called from. This parameter must be the value of `wherefrom` passed to the user-written callback.

`double nodeest`

A double that indicates the value of the node estimate for the node to be created with this branch. The node estimate is used to select nodes from the branch & cut tree with certain values of the `NodeSel` parameter.

`int varcnt`

An integer that indicates the number of bound changes that are specified in the arrays `varind`, `varlu`, and `varbd`.

`const int *varind`

Together with `varlu` and `varbd`, this array defines the bound changes for the branch. The entry `varind[i]` is the index for the variable.

`const char *varlu`

Together with `varind` and `varbd`, this array defines the bound changes for the branch. The entry `varlu[i]` is one of three possible values indicating which bound to change:

- `L` for lower bound,
- `U` for upper bound, or
- `B` for both bounds.

`const int *varbd`

Together with `varind` and `varlu`, this array defines the bound changes for the branch. The entry `varbd[i]` indicates the new value of the bound.

`int rcnt`

An integer that indicates the number of constraints for the branch.

`int nzcnt`

An integer that indicates the number of nonzero constraint coefficients for the branch. This specifies the length of the arrays `rmatind` and `rmatval`.

`const double *rhs`

An array of length `rcnt` containing the right-hand side term for each constraint for the branch.

`const char *sense`

An array of length `rcnt` containing the sense of each constraint to be added for the branch.

```
sense[i] = 'L' ≤constraint
sense[i] = 'E' = constraint
sense[i] = 'G' ≥ constraint
```

`const int *rmatbeg`

`const int *rmatind`

`const double *rmatval`

Arrays that describe the constraints for the branch. The format is similar to the format used to describe the constraint matrix in the routine `CPXaddrows()`. Every row must be stored in sequential locations in this array from position `rmatbeg[i]` to `rmatbeg[i+1]-1` (or from `rmatbeg[i]` to `nzcnt -1` if `i=rcnt-1`).

Each entry, `rmatind[i]`, indicates the column index of the corresponding coefficient, `rmatval[i]`. All rows must be contiguous, and `rmatbeg[0]` must be `0`.

`void *userhandle`

A pointer to user private data that should be associated with the node created by this branch. May be `NULL`.

`int *seqnum_p`

A pointer to an integer that, on return, will contain the sequence number that CPLEX has assigned to the node created from this branch. The sequence number may be used to select this node in later calls to the node callback.

CPXbtran

Usage

Advanced

Description

The routine `CPXbtran()` solves $x^T\boldsymbol{B} = y^T$ and puts the answer in y. $\boldsymbol{B}$ is the basis matrix.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXbtran (CPXCENVptr env,
              CPXCLPptr lp,
              double *y);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to the CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`double *y`

An array that holds the right-hand side vector on input and the solution vector on output. The array must be of length at least equal to the number of rows in the LP problem object.

CPXcheckax

Usage Advanced

Description The routine `CPXcheckax()` finds the L_∞ norm of $\boldsymbol{A}x$ - b. That is, this routine checks for numerical (roundoff) error in the computation of x (the resident solution) by putting it into that formula and determining which row has the maximum error from zero. This routine also returns, in one of its arguments, the index of the row with the maximum error from zero.

To get the L_∞ norm for the scaled problem, set the parameter `scalrimtype = 1`.

Return Value If successful, the routine returns the L_∞ norm of $\boldsymbol{A}x$ - b, where x is the resident solution. If no such solution exists, `-1.0` is returned.

Synopsis

```
double CPXcheckax (CPXCENVptr env,
                   CPXCLPptr lp,
                   int *imax_p,
                   int scalrimtype);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to the CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int *imax_p`

A pointer to the index of the row with the maximum absolute value in $\boldsymbol{A}x$ - b. If no solution exists, `*imax_p` is set to `-1`.

`int scalrimtype`

An integer that indicates the type of scaling to be applied to the returned L_∞ norm. When this parameter is equal to `0` (zero), the returned L_∞ norm will be unscaled. Otherwise, the L_∞ norm has the same scaling as that applied to the problem currently in memory.

CPXcheckpib

Usage

Advanced

Description

The routine `CPXcheckpib()` finds the L_∞ norm of $c_B{}^T - \pi^T \mathbf{B}$, where π represents dual solution values and **B** represents the basis. That is, this routine checks for numerical (roundoff) error in the computation of π by putting π into the equation that defines it and then returning the value of the maximum deviation from zero of the elements of the resulting residual vector. This routine also returns, in one of its arguments, the index of the basic variable corresponding to this maximum.

To get the L_∞ norm for the scaled problem, set the parameter `scalrimtype = 1`.

Return Value

If successful, this routine returns the L_∞ norm of $c_B{}^T - \pi^T \mathbf{B}$. If a basic solution does not exist, `-1.0` is returned.

Synopsis

```
double CPXcheckpib (CPXCENVptr env,
                    CPXCLPptr lp,
                    int *ijmax_p,
                    int scalrimtype);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to the CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int *ijmax_p`

A pointer to the row or column with the maximum absolute value in $c_B{}^T - \pi^T \mathbf{B}$. If `*ijmax_p` corresponds to a row numbered `rowindex` (either a slack row or a ranged row), `*ijmax_p` is `-1 - rowindex`. If no solution exists, `*ijmax_p` is set to a large integer.

`int scalrimtype`

An integer that indicates the type of scaling to be applied to the returned L_∞ norm. When this parameter is equal to `0` (zero), the returned L_∞ norm is unscaled. Otherwise, the L_∞ norm has the same scaling as that applied to the problem currently in memory.

CPXcopybasednorms

Usage Advanced

Description The routine `CPXcopybasednorms()` works in conjunction with the routine `CPXgetbasednorms()`. `CPXcopybasednorms()` copies the values in the arrays `cstat`, `rstat`, and `dnorm`, as returned by `CPXgetbasednorms()`, into a specified problem object.

Each of the arrays `cstat`, `rstat`, and `dnorm` must be non `NULL`. Only data returned by `CPXgetbasednorms()` should be copied by `CPXcopybasednorms()`. (Other details of `cstat`, `rstat`, and `dnorm` are not documented.)

> ***Important:*** *The routine* `CPXcopybasednorms()` *should be called only if the return values of* `CPXgetnumrows()` *and* `CPXgetnumcols()` *have not changed since the companion call to* `CPXgetbasednorms()`*. If either of these values has increased since that companion call, a memory violation may occur. If one of those values has decreased, the call will be safe, but its meaning will be undefined.*

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
double CPXcopybasednorms (CPXCENVptr env,
                          CPXLPptr lp,
                          const int *cstat,
                          const int *rstat,
                          const double *dnorm);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to the CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`const int *cstat`

An array containing the basis status of the columns in the constraint matrix returned by a call to `CPXgetbasednorms()`. The length of the allocated array must be at least the value returned by `CPXgetnumcols()`.

`const int *rstat`

An array containing the basis status of the rows in the constraint matrix returned by a call to `CPXgetbasednorms()`. The length of the allocated array must be at least the value returned by `CPXgetnumrows()`.

`const double *dnorm`

An array containing the dual steepest-edge norms returned by a call to `CPXgetbasednorms()`. The length of the allocated array must be at least the value returned by `CPXgetnumrows()`.

See Also

CPXgetbasednorms()

CPXgetnumcols(), CPXgetnumrows() (ILOG CPLEX Reference Manual)

CPXcopydnorms

Usage Advanced

Description The routine `CPXcopydnorms()` copies the dual steepest-edge norms to the specified LP problem object. The argument `head` is an array of column or row indices corresponding to the array of norms. Column indices are indexed with nonnegative values. Row indices are indexed with negative values offset by 1 (one). For example, if `head[0] = -5`, then `norm[0]` is associated with `row 4`.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXcopydnorms (CPXCENVptr env,
                   CPXLPptr lp,
                   const double *norm,
                   const int *head
                   int len);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to the CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`const double *norm`

An array containing values to be used in a subsequent call to `CPXdualopt()`, with a setting of `CPX_PARAM_DGRADIENT` different from `1` (one), as the initial values for the dual steepest-edge norms of the corresponding basic variables specified in `head[]`. The array must be of length at least equal to the value of the argument `len`.

If any indices in `head[]` are not basic, the corresponding values in `norm[]` are ignored.

`const int *head`

An array containing the indices of the basic variables for which norms have been specified in `norm[]`. The array must be of length at least equal to the value of the argument `len`.

`int len`

An integer that indicates the number of entries in `norm[]` and `head[]`.

See Also *CPXcopypnorms(), CPXgetdnorms()*

CPXcopypartialbase

Description

The routine `CPXcopypartialbase()` is used to copy a a partial basis into an LP problem object. Basis statuses do not need to be specified for every variable or slack/surplus/artificial variable. If the status of a variable is not specified, it is made non-basic at lower bound if the lower bound is finite, otherwise non-basic at upper bound if the upper bound is finite, otherwise non-basic at 0.0. If the status of a slack/surplus/ artificial variable is not specified, it is made basic.

Return Value

This routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXcopypartialbase (CPXCENVptr env,
                        CPXLPptr lp,
                        int ccnt,
                        const int *cindices,
                        const int *cstat,
                        int rcnt,
                        const int *rindices,
                        const int *rstat);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int ccnt`

An integer that indicates the number of variable/column statuses specified, and is the length of the `cindices` and `cstat` arrays.

`const int *cindices`

An array of length `ccnt` that contains the indices of the variables for which statuses are being specified.

`const int *cstat`

An array of length `ccnt` where the i^{th} entry contains the status for variable `cindices[i]`.

Values for `cstat[i]`:

`CPX_AT_LOWER`	`0`	variable at lower bound
`CPX_BASIC`	`1`	variable is basic

`CPX_AT_UPPER`	2	variable at upper bound
`CPX_FREE_SUPER`	3	variable free and non-basic

`int rcnt`

An integer that indicates the number of slack/surplus/artificial statuses specified, and is the length of the `rindices` and `rstat` arrays.

`const int *rindices`

An array of length `rcnt` that contains the indices of the slack/surplus/artificials for which statuses are being specified.

`const int *rstat`

An array of length `rcnt` where the i^{th} entry contains the status for slack/surplus/artificial `rindices[i]`. For rows other than ranged rows, the array element `rstat[i]` has the following meaning:

`CPX_AT_LOWER`	0	associated slack variable non-basic at value 0.0
`CPX_BASIC`	1	associated slack/surplus/artificial variable basic

For ranged rows, the array element `rstat[i]` has the following meaning:

`CPX_AT_LOWER`	0	associated slack variable non-basic at its lower bound
`CPX_BASIC`	1	associated slack variable basic
`CPX_AT_UPPER`	2	associated slack variable non-basic at its upper bound

Example

```
status = CPXcopypartialbase (env, lp, ccnt, colind, colstat,
                             rcnt, rowind, rowstat);
```

CPXcopypnorms

Usage Advanced

Description The routine `CPXcopypnorms()` copies the primal steepest-edge norms to the specified LP problem object.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXcopypnorms (CPXENVptr env,
                   CPXLPptr lp,
                   double *cnorm,
                   double *rnorm,
                   int len);
```

Arguments

`CPXENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`double *cnorm`

An array containing values to be used in a subsequent call to `CPXprimopt()`, with a setting of `CPX_PARAM_PGRADIENT` of 2 or 3, as the initial values for the primal steepest-edge norms of the first `len` columns in the LP problem object. The array must be of length at least equal to the value of the argument `len`.

`double *rnorm`

An array containing values to be used in a subsequent call to `CPXprimopt()` with a setting of `CPX_PARAM_PGRADIENT` of 2 or 3, as the initial values for the primal steepest-edge norms of the slacks and ranged variables that are nonbasic. The array must be of length at least equal to the number of rows in the LP problem object.

`int len`

An integer that indicates the number of entries in the array `cnorm[]`.

See Also *CPXcopydnorms(), CPXgetpnorms()*

CPXcopyprotected

Usage Advanced

Description The routine `CPXcopyprotected()` is used to specify a set of variables that should not be substituted out of the problem. If presolve can fix a variable to a value, it is removed, even if it is specified in the protected list.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXcopyprotected (CPXCENVptr env,
                      CPXLPptr lp,
                      int cnt,
                      const int *indices);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int cnt`

The number of variables to be protected.

`const int *indices`

An array of length `cnt` containing the column indices of variables to be protected from being substituted out.

Example

```
status = CPXcopyprotected (env, lp, cnt, indices);
```

CPXcrushform

Usage Advanced

Description The routine `CPXcrushform()` crushes a linear formula of the original problem to a linear formula of the presolved problem.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXcrushform (CPXCENVptr env,
                  CPXCLPptr lp,
                  int len,
                  const int *ind,
                  const double *val,
                  int *plen_p,
                  double *poffset_p,
                  int *pind,
                  double *pval);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int len`

The number of entries in the arrays `ind` and `val`.

`const int *ind`
`const double *val`

The linear formula in terms of the original problem. Each entry, `ind[i]`, indicates the column index of the corresponding coefficient, `val[i]`.

`int *plen_p`

A pointer to an integer to receive the number of nonzero coefficients, that is, the true length of the arrays `pind` and `pval`.

`double *poffset_p`

A pointer to a double to contain the value of the linear formula corresponding to variables that have been removed in the presolved problem.

```
int *pind
double *pval
```

The linear formula in terms of the presolved problem. Each entry, `pind[i]`, indicates the column index in the presolved problem of the corresponding coefficient, `pval[i]`. The arrays `pind` and `pval` must be of length at least the number of columns in the presolved LP problem object.

Example

```
status = CPXcrushform (env, lp, len, ind, val,
                       &plen, &poffset, pind, pval);
```

CPXcrushpi

Usage

Advanced

Description

The routine `CPXcrushpi()` crushes a dual solution for the original problem to a dual solution for the presolved problem.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXcrushpi (CPXCENVptr env,
                CPXCLPptr lp,
                const double *pi,
                double *prepi);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`const double *pi`

An array that contains dual solution (`pi`) values for the original problem, as returned by routines such as `CPXgetpi()` or `CPXsolution()`. The array must be of length at least the number of rows in the LP problem object.

`double *prepi`

An array to receive dual values corresponding to the presolved problem. The array must be of length at least the number of rows in the presolved problem object.

Example

```
status = CPXcrushpi (env, lp, origpi, reducepi);
```

CPXcrushx

Usage
Advanced

Description
The routine `CPXcrushx()` crushes a solution for the original problem to a solution for the presolved problem.

Return Value
The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis
```
int CPXcrushx (CPXCENVptr env,
               CPXCLPptr lp,
               const double *x,
               double *prex);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`const double *x`

An array that contains primal solution (x) values for the original problem, as returned by routines such as `CPXgetx()` or `CPXsolution()`. The array must be of length at least the number of columns in the problem object.

`double *prex`

An array to receive the primal values corresponding to the presolved problem. The array must be of length at least the number of columns in the presolved problem object.

Example
```
status = CPXcrushx (env, lp, origx, reducex);
```

See Also
Example `admipex6.c` *in the advanced examples directory.*

CPXcutcallbackadd

Usage Mixed Integer Users Only

Description The routine `CPXcutcallbackadd()` adds cuts during MIP branch & cut. This routine may be called only from within user-written cut callbacks; thus it may be called only when the value of its `wherefrom` argument is `CPX_CALLBACK_MIP_CUT`.

The cut may be for the original problem if the parameter `CPX_PARAM_MIPCBREDLP` was set to `CPX_OFF` before the call to `CPXmipopt()` that calls the callback. Otherwise, the cut is used on the presolved problem.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

One possible error is indicated by the symbolic constant `CPXERR_NO_SPACE`. That error occurs when the number of cuts added reaches the maximum allowed, as set by the parameter `CPX_PARAM_CUTSFACTOR`.

Synopsis

```
int CPXcutcallbackadd (CPXCENVptr env,
                       void *cbdata,
                       int wherefrom,
                       int *nzcnt,
                       double rhs,
                       int sense,
                       const int *cutind,
                       const double *cutval);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter *must* be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value that indicates where the user-written callback was called from. This parameter *must* be the value of `wherefrom` passed to the user-written callback.

`int *nzcnt`

An integer value that indicates the number of coefficients in the cut, or equivalently, the length of the arrays `cutind` and `cutval`.

`double rhs`

A double value that indicates the value of the right-hand side of the cut.

`int sense`

An integer value that indicates the sense of the cut.

`const int *cutind`

An array containing the column indices of cut coefficients.

`const double *cutval`

An array containing the values of cut coefficients.

Example

```
status = CPXcutcallbackadd (env,
                            cbdata,
                            wherefrom,
                            mynzcnt,
                            myrhs,
                            'L',
                            mycutind,
                            mycutval);
```

See Also

The example `admipex5.c`

CPXgetcutcallbackfunc(), CPXsetcutcallbackfunc()

CPXcutcallbackaddlocal

Usage

Mixed Integer Users Only

Description

The routine `CPXcutcallbackaddlocal()` adds local cuts during MIP branch & cut. A local cut is one that applies to the current nodes and the subtree rooted at this node. Global cuts, that is, cuts that apply throughout the branch & cut tree, are added with the routine `CPXcutcallbackadd()`. This routine may be called only from within user-written cut callbacks; thus it may be called only when the value of its `wherefrom` argument is `CPX_CALLBACK_MIP_CUT`.

The cut may be for the original problem if the parameter `CPX_PARAM_MIPCBREDLP` was set to `CPX_OFF` before the call to `CPXmipopt()` that calls the callback. Otherwise, the cut is used on the presolved problem.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXcutcallbackaddclocal (CPXCENVptr env,
                             void *cbdata,
                             int wherefrom,
                             int *nzcnt,
                             double rhs,
                             int sense,
                             const int *cutind,
                             const double *cutval);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter *must* be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value that indicates where the user-written callback was called from. This parameter *must* be the value of `wherefrom` passed to the user-written callback.

`int *nzcnt`

An integer value that indicates the number of coefficients in the cut, or equivalently, the length of the arrays `cutind` and `cutval`.

`double rhs`

A double value that indicates the value of the right-hand side of the cut.

`int sense`

An integer value that indicates the sense of the cut.

`const int *cutind`

An array containing the column indices of cut coefficients.

`const double *cutval`

An array containing the values of cut coefficients.

Example

```
status = CPXcutcallbackaddlocal (env,
                                 cbdata,
                                 wherefrom,
                                 mynzcnt,
                                 myrhs,
                                 'L',
                                 mycutind,
                                 mycutval);
```

See Also

CPXcutcallbackadd(), CPXgetcutcallbackfunc(), CPXsetcutcallbackfunc()

CPXdjfrompi

Usage Advanced

Description The routine `CPXdjfrompi()` computes an array of reduced costs from an array of dual values. This routine is for linear programs. Use `CPXqpdjfrompi()` for quadratic programs.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXdjfrompi (CPXCENVptr env,
                 CPXCLPptr lp,
                 const double *pi,
                 double *dj);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`const double *pi`

An array that contains dual solution (`pi`) values for the problem, as returned by routines such as `CPXuncrushpi()` and `CPXcrushpi()`. The array must be of length at least the number of rows in the problem object.

`double *dj`

An array to receive the reduced cost values computed from the `pi` values for the problem object. The array must be of length at least the number of columns in the problem object.

Example

```
status = CPXdjfrompi (env, lp, pi, dj);
```

CPXdualfarkas

Usage

Advanced

Description

The routine `CPXdualfarkas()` assumes that there is a resident solution as produced by a call to `CPXdualopt()` and that the status of this solution as returned by `CPXgetstat()` is `CPX_UNBOUNDED`.

The values returned in the array y[] have the following interpretation. For the i^{th} constraint, if that constraint is a less-than-or-equal-to constraint, $y[i] \leq 0$ holds; if that constraint is a greater-than-or-equal-to constraint, $y[i] \geq 0$ holds. Thus, where `b` is the right-hand-side vector for the given linear program, A is the constraint matrix, and x denotes the vector of variables, y may be used to derive the following valid inequality:

$$y^T\boldsymbol{A}\, x \geq y^T b$$

Here y is being interpreted as a column vector, and y^T denotes the transpose of y.

The real point of computing y is the following. Suppose we define a vector z of dimension equal to the dimension of x and having the following value for entries

$z_j = u_j$ where $y^T A_j > 0$, and

$z_j = l_j$ where $y^T\boldsymbol{A}_j < 0$,

where $\boldsymbol{A}_j$ denotes the column of $\boldsymbol{A}$ corresponding to x_j, u_j the given upper bound on x_j, and l_j is the specified lower bound. (z_j is arbitrary if $y^T\boldsymbol{A}_j = 0$.) Then y and z will satisfy

$$y^T b - y^T\boldsymbol{A}\, z > 0.$$

This last inequality contradicts the validity of $y^T\boldsymbol{A}\, x \geq y^T b$, and hence shows that the given linear program is infeasible. The quantity `*proof_p` is set equal to $y^T b - y^T\boldsymbol{A}\, z$. Thus, `*proof_p` in some sense denotes the degree of infeasibility.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXdualfarkas (CPXCENVptr env,
                   CPXCLPptr lp,
                   double *y,
                   double *proof_p);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`double *y`

An array of doubles of length at least equal to the number of rows in the problem.

`double *proof_p`

A pointer to a double. The parameter `proof_p` is allowed to have the value `NULL`.

CPXfreelazyconstraints

Usage Mixed Integer Users Only

Description The routine `CPXfreelazyconstraints()` is used to clear the list of lazy constraints that have been previously specified through calls to `CPXaddlazyconstraints()`.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXfreelazyconstraints (CPXCENVptr env,
                            CPXLPptr lp);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

Example

```
status = CPXfreelazyconstraints (env, lp);
```

CPXfreepresolve

Usage

Advanced

Description

The routine `CPXfreepresolve()` frees the presolved problem from the LP problem object. Under the default setting of `CPX_PARAM_REDUCE`, the presolved problem is freed when an optimal solution is found. It is not freed when `CPX_PARAM_REDUCE` is set to `CPX_PREREDUCE_PRIMALONLY` (1) or `CPX_PREREDUCE_DUALONLY` (2), so the routine `CPXfreepresolve()` can be used to free it manually.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXfreepresolve (CPXCENVptr env,
                     CPXLPptr lp);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

Example

```
status = CPXfreepresolve (env, lp);
```

CPXfreeusercuts

Usage Mixed Integer Users Only

Description The routine `CPXfreeusercuts()` is used to clear the list of user cuts that have been previously specified through calls to `CPXaddusercuts()`.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXfreeusercuts (CPXCENVptr env,
                     CPXLPptr lp);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

Example

```
status = CPXfreeusercuts (env, lp);
```

CPXftran

Usage Advanced

Description The routine `CPXftran()` solves $\boldsymbol{B}y = x$ and puts the answer in the vector x, where $\boldsymbol{B}$ is the basis matrix.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXftran (CPXCENVptr env,
              CPXCLPptr lp,
              double *x);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`double *x`

An array that holds the right-hand side vector on input and the solution vector on output. The array must be of length at least equal to the number of rows in the LP problem object.

CPXgetbasednorms

Usage Advanced

Description The routine `CPXgetbasednorms()` works in conjunction with the routine `CPXcopybasednorms()`. `CPXgetbasednorms()` retrieves the resident basis and dual norms from a specified problem object.

Each of the arrays `cstat`, `rstat`, and `dnorm` must be non `NULL`. That is, each of these arrays must be allocated. The allocated size of `cstat` is assumed by this routine to be at least the number returned by `CPXgetnumcols()`. The allocated size of `rstat` and `dnorm` are assumed to be at least the number returned by `CPXgetnumrows()`. (Other details of `cstat`, `rstat`, and `dnorm` are not documented.)

Success, Failure

If this routine succeeds, `cstat` and `rstat` contain information about the resident basis, and `dnorm` contains the dual steepest-edge norms. If there is no basis, or if there is no set of dual steepest-edge norms, this routine returns an error code. The returned data are intended solely for use by `CPXcopybasednorms()`.

Example

For example, if a given LP has just been successfully solved by the ILOG CPLEX Callable Library optimizer `CPXdualopt()` with the dual pricing option `CPX_PARAM_DPRIIND` set to `CPX_DPRIIND_STEEP`, `CPX_DPRIIND_FULLSTEEP`, or `CPX_DPRIIND_STEEPQSTART`, then a call to `CPXgetbasednorms()` should succeed. (That optimizer and those pricing options are documented in the *ILOG CPLEX Reference Manual*, and their use is illustrated in the *ILOG CPLEX User's Manual*.)

Motivation

When the ILOG CPLEX Callable Library optimizer `CPXdualopt()` is called to solve a problem with the dual pricing option `CPX_PARAM_DPRIIND` set to `CPX_DPRIIND_STEEP or CPX_DPRIIND_FULLSTEEP`, there must be values of appropriate dual norms available before the optimizer can begin. If these norms are not already resident, they must be computed, ant that computation may be expensive. The functions `CPXgetbasednorms()` and `CPXcopybasednorms()` can, in some cases, avoid that expense. Suppose, for example, that in some application an LP is solved by `CPXdualopt()` with one of those pricing settings. After the solution of the LP, some intermediate optimizations are carried out on the same LP, and those subsequent optimizations are in turn followed by some changes to the LP, and a re-solve. In such a case, copying the basis and norms that were resident before the intermediate solves, back into ILOG CPLEX data structures can greatly increase the speed of the re-solve.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetbasednorms (CPXCENVptr env,
                      CPXCLPptr lp,
                      int *cstat,
                      int *rstat,
                      double *dnorm);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to the CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int *cstat`

An array containing the basis status of the columns in the constraint matrix. The length of the allocated array is at least the value returned by `CPXgetnumcols()`.

`int *rstat`

An array containing the basis status of the rows in the constraint matrix. The length of the allocated array is at least the value returned by `CPXgetnumrows()`.

`double *dnorm`

An array containing the dual steepest-edge norms. The length of the allocated array is at least the value returned by `CPXgetnumrows()`.

See Also

CPXcopybasednorms()

CPXgetnumcols(), CPXgetnumrows() (ILOG CPLEX Reference Manual)

CPXgetbhead

Usage Advanced

Description The routine `CPXgetbhead()` returns the basis header; it gives the negative value minus one of all row indices of slacks.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetbhead (CPXCENVptr env,
                 CPXCLPptr lp,
                 int *head,
                 double *x);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int *head`

An array containing the indices of the variables in the resident basis, where basic slacks are specified by the negative of the corresponding row index minus 1 (one); that is, `-rowindex - 1`. The array must be of length at least equal to the number of rows in the LP problem object.

`double *x`

An array containing the values of the basic variables in the order specified by `head[]`. The array must be of length at least equal to number of rows in the LP problem object.

CPXgetbranchcallbackfunc

Usage

Mixed Integer Users Only

Description

The routine `CPXgetbranchcallbackfunc()` accesses the user-written callback routine to be called during MIP optimization after a branch has been selected but before the branch is carried out. ILOG CPLEX uses the callback routine to change its branch selection.

For documentation of callback arguments, see the routine `CPXsetbranchcallbackfunc()`.

Return Value

This routine does not return a result.

Synopsis

```
void CPXgetbranchcallbackfunc(CPXCENVptr env,
                              int (CPXPUBLIC *branchcallback)
                                     (CPXCENVptr env,
                                      void *cbdata,
                                      int wherefrom,
                                      void *cbhandle,
                                      int type,
                                      int sos,
                                      int nodecnt,
                                      int bdcnt,
                                      double *nodeest,
                                      int *nodebeg,
                                      int *indices,
                                      char *lu,
                                      int *bd,
                                      int *useraction_p),
                             void *cbhandle);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`int (CPXPUBLIC *branchcallback)`

The address of the pointer to the current user-written branch callback. If no callback has been set, the returned pointer evaluates to `NULL`.

`void *cbhandle`

The address of a variable to hold the user's private pointer.

Example

```
CPXgetbranchcallbackfunc(env, &current_callback,
                         &current_handle);
```

See Also *CPXsetbranchcallbackfunc(), Advanced MIP Control Interface*

CPXgetcallbackctype

Usage

Mixed Integer Users Only

Description

The routine `CPXgetcallbackctype()` is used to get the ctypes for the MIP problem from within a user-written callback during MIP optimization. The values are from the original problem if `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF`, otherwise they are from the presolved problem.

This routine may be called only when the value of the `wherefrom` argument is one of `CPX_CALLBACK_MIP`, `CPX_CALLBACK_MIP_BRANCH`, `CPX_CALLBACK_MIP_INCUMBENT`, `CPX_CALLBACK_MIP_NODE`, `CPX_CALLBACK_MIP_HEURISTIC`, `CPX_CALLBACK_MIP_SOLVE`, or `CPX_CALLBACK_MIP_CUT`.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetcallbackctype (CPCXENVptr env,
                         void *cbdata,
                         int wherefrom,
                         char *xctype,
                         int begin,
                         int end);
```

Arguments

`CPCXENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter must be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating from where the user-written callback was called. The parameter must be the value of `wherefrom` passed to the user-written callback.

`char *xctype`

An array where the ctype values for the MIP problem will be returned. The array must be of length at least (`end - begin + 1`). If successful, `xctype[0]` through `xctype[end-begin]` contain the variable types.

`int begin`

An integer indicating the beginning of the range of ctype values to be returned.

`int end`

An integer indicating the end of the range of ctype values to be returned.

Example

```
status = CPXgetcallbackctype (env, cbdata, wherefrom,
                              prectype, 0, precols-1);
```

CPXgetcallbackgloballb

Usage

Mixed Integer Users Only

Description

The routine `CPXgetcallbackgloballb()` is used to get the best known global lower bound values during MIP optimization from within a user-written callback. The values are from the original problem if `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF`, otherwise they are from the presolved problem.

This routine may be called only when the value of the `wherefrom` argument is one of `CPX_CALLBACK_MIP`, `CPX_CALLBACK_MIP_BRANCH`, `CPX_CALLBACK_MIP_INCUMBENT`, `CPX_CALLBACK_MIP_NODE`, `CPX_CALLBACK_MIP_HEURISTIC`, `CPX_CALLBACK_MIP_SOLVE`, or `CPX_CALLBACK_MIP_CUT`.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetcallbackgloballb (CPXCENVptr env,
                            void *cbdata,
                            int wherefrom,
                            double *lb,
                            int begin,
                            int end);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter must be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating from where the user-written callback was called. The parameter must be the value of `wherefrom` passed to the user-written callback.

`double *lb`

An array to receive the values of the global lower bound values. This array must be of length at least (`end - begin + 1`). If successful, `lb[0]` through `lb[end-begin]` contain the global lower bound values.

`int begin`

An integer indicating the beginning of the range of lower bound values to be returned.

`int end`

An integer indicating the end of the range of lower bound values to be returned.

Example

```
status = CPXgetcallbackgloballb (env, cbdata, wherefrom,
                                 glb, 0, cols-1);
```

CPXgetcallbackglobalub

Usage

Mixed Integer Users Only

Description

The routine `CPXgetcallbackglobalub()` is used to get the best known global upper bound values during MIP optimization from within a user-written callback. The values are from the original problem if `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF`, otherwise they are from the presolved problem.

This routine may be called only when the value of the `wherefrom` argument is one of `CPX_CALLBACK_MIP`, `CPX_CALLBACK_MIP_BRANCH`, `CPX_CALLBACK_MIP_INCUMBENT`, `CPX_CALLBACK_MIP_NODE`, `CPX_CALLBACK_MIP_HEURISTIC`, `CPX_CALLBACK_MIP_SOLVE`, or `CPX_CALLBACK_MIP_CUT`.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetcallbackglobalub (CPXCENVptr env,
                            void *cbdata,
                            int wherefrom,
                            double *ub,
                            int begin,
                            int end);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter must be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating from where the user-written callback was called. The parameter must be the value of `wherefrom` passed to the user-written callback.

`double *ub`

An array to receive the values of the global upper bound values. This array must be of length at least (`end - begin + 1`). If successful, `ub[0]` through `ub[end-begin]` contain the global upper bound values.

`int begin`

An integer indicating the beginning of the range of upper bound values to be returned.

`int end`

An integer indicating the end of the range of upper bound values to be returned.

Example

```
status = CPXgetcallbackglobalub (env, cbdata, wherefrom,
                                 gub, 0, cols-1);
```

CPXgetcallbackincumbent

Usage

Mixed Integer Users Only

Description

The routine `CPXgetcallbackincumbent()` is used to get the incumbent values during MIP optimization from within a user-written callback. The values are from the original problem if `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF`, otherwise they are from the presolved problem.

This routine may be called only when the value of the `wherefrom` argument is one of `CPX_CALLBACK_MIP`, `CPX_CALLBACK_MIP_BRANCH`, `CPX_CALLBACK_MIP_INCUMBENT`, `CPX_CALLBACK_MIP_NODE`, `CPX_CALLBACK_MIP_HEURISTIC`, `CPX_CALLBACK_MIP_SOLVE`, or `CPX_CALLBACK_MIP_CUT`.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetcallbackincumbent (CPXCENVptr env,
                             void *cbdata,
                             int wherefrom,
                             double *x,
                             int begin,
                             int end);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter must be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating from where the user-written callback was called. The parameter must be the value of `wherefrom` passed to the user-written callback.

`double *x`

An array to receive the values of the incumbent (best available) integer solution. This array must be of length at least (`end - begin + 1`). If successful, `x[0]` through `x[end-begin]` contain the incumbent values.

`int begin`

An integer indicating the beginning of the range of incumbent values to be returned.

`int end`

An integer indicating the end of the range of incumbent values to be returned.

Example

```
status = CPXgetcallbackincumbent (env, cbdata, wherefrom,
                                  bestx, 0, cols-1);
```

CPXgetcallbacklp

Usage

Mixed Integer Users Only

Description

The routine `CPXgetcallbacklp()` is used to get the pointer to the MIP problem that is in use when the user-written callback function is called. It is the original MIP if `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF`, otherwise it is the presolved MIP. In contrast, the function `CPXgetcallbacknodelp()` returns a pointer to the node subproblem, which is an LP. Generally, this pointer may be used only in CPLEX Callable Library query routines, such as `CPXsolution()` or `CPXgetrows()`.

This routine may be called only when the value of the `wherefrom` argument is one of `CPX_CALLBACK_MIP`, `CPX_CALLBACK_MIP_BRANCH`, `CPX_CALLBACK_MIP_INCUMBENT`, `CPX_CALLBACK_MIP_NODE`, `CPX_CALLBACK_MIP_HEURISTIC`, `CPX_CALLBACK_MIP_SOLVE`, or `CPX_CALLBACK_MIP_CUT`.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetcallbacklp (CPXCENVptr env,
                      void *cbdata,
                      int wherefrom,
                      CPXCLPptr *lp_p);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter must be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating from where the user-written callback was called. The parameter must be the value of `wherefrom` passed to the user-written callback.

`CPXCLPptr *lp_p`

A pointer to a variable of type `CPXLPptr` to receive the pointer to the LP problem object, which is a MIP.

Example

```
status = CPXgetcallbacklp (env, cbdata, wherefrom, &origlp);
```

See Also

Examples `admipex1.c`, `admipex2.c`, *and* `admipex3.c` *in the advanced examples directory*

CPXgetcallbacknodeinfo

Usage Mixed Integer Users Only

Description The routine `CPXgetcallbacknodeinfo()` is called from within user-written callbacks during a MIP optimization and accesses information about nodes. When the `wherefrom` argument is `CPX_CALLBACK_MIP_NODE`, a node with any `nodeindex` value can be queried. When the `wherefrom` argument is any one of `CPX_CALLBACK_MIP_CUT`, `CPX_CALLBACK_MIP_INCUMBENT`, `CPX_CALLBACK_MIP_HEURISTIC`, or `CPX_CALLBACK_MIP_BRANCH`, only the current node can be queried. This is done by specifying a `nodeindex` value of 0. Other values of the `wherefrom` argument are invalid for this routine; an invalid `nodeindex` value or `wherefrom` argument value will result in an error return value.

> ***Notes:*** *The values returned for* `CPX_CALLBACK_INFO_NODE_SIINF` *and* `CPX_CALLBACK_INFO_NODE_NIINF` *for the current node are the values that applied to the node when it was stored and thus before the branch was solved. As a result, these values should not be used to assess the feasibility of the node.*
>
> *This routine cannot retrieve information about nodes that have been moved to node files. For more information about node files, see the* ILOG CPLEX User's Manual. *If the argument* `nodeindex` *refers to a node in a node file,* `CPXgetnodecallbackinfo()` *returns the value* `CPXERR_NODE_ON_DISK`. *Nodes still in memory have the lowest index numbers so a user can loop through the nodes until* `CPXgetcallbacknodeinfo()` *returns an error, and then exit the loop.*

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

The return value `CPXERR_NODE_ON_DISK` indicates an attempt to access a node currently located in a node file on disk.

Synopsis

```
int CPXgetcallbacknodeinfo (CPXCENVptr env,
                            void *cbdata,
                            int wherefrom,
                            int nodeindex,
                            int whichinfo,
                            void *result_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter *must* be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating where the user-written callback was called from. This parameter *must* be the value of `wherefrom` passed to the user-written callback.

`int nodeindex`

The index of the node for which information is requested. Nodes are indexed from `0` (zero) to `(nodecount - 1)` where `nodecount` is obtained from the callback information function `CPXgetcallbackinfo()`, with a `whichinfo` value of `CPX_CALLBACK_INFO_NODES_LEFT`.

`int whichinfo`

An integer indicating which information is requested. Table 1 summarizes possible values.

Table 1 *Information Requested for a User-Written Node Callback*

Symbolic Constant	C Type	Meaning
CPX_CALLBACK_INFO_NODE_SIINF	double	sum of integer infeasibilities
CPX_CALLBACK_INFO_NODE_NIINF	int	number of integer infeasibilities
CPX_CALLBACK_INFO_NODE_ESTIMATE	double	estimated integer objective
CPX_CALLBACK_INFO_NODE_DEPTH	int	depth of node in branch & cut tree
CPX_CALLBACK_INFO_NODE_OBJVAL	double	objective value of LP subproblem
CPX_CALLBACK_INFO_NODE_TYPE	int	type of branch at this node; see Table 2
CPX_CALLBACK_INFO_NODE_VAR	int	for nodes of type CPX_TYPE_VAR, the branching variable for this node; for SOS-type branches, the rightmost variable in left subset
CPX_CALLBACK_INFO_NODE_SOS	int	the number of the SOS used in branching; -1 if none used
CPX_CALLBACK_INFO_NODE_SEQNUM	int	sequence number of the node
CPX_CALLBACK_INFO_NODE_NODENUM	int	node index of the node

Table 2 summarizes possible values returned when the type of information requested is branch type (that is, `whichinfo` = `CPX_CALLBACK_INFO_NODE_TYPE`).

Table 2 *Branch Types Returned when `whichinfo = CPX_CALLBACK_INFO_NODE_TYPE`*

Symbolic Constant	Value	Branch Type
CPX_TYPE_VAR	'0'	variable branch
CPX_TYPE_SOS1	'1'	SOS1 branch
CPX_TYPE_SOS2	'2'	SOS2 branch
CPX_TYPE_USER	'X'	user-defined

`void *result_p`

A generic pointer to a variable of type `double` or `int`, representing the value returned by `whichinfo`. (The column "C Type" in Table 1 indicates the type of various values returned by `whichinfo`.)

Example

```
status = CPXgetnodecallbackinfo(env,
                                curlp,
                                wherefrom,
                                0,
                                CPX_CALLBACK_INFO_NODE_NIINF,
                                &numiinf);
```

See Also

CPXgetcallbackinfo() (ILOG CPLEX Reference Manual), CPXgetcallbackseqinfo(), Advanced MIP Control Interface

CPXgetcallbacknodeintfeas

Usage

Mixed Integer Users Only

Description

The routine `CPXgetcallbacknodeintfeas()` is used to get indicators for each variable of whether or not the variable is integer feasible in the node subproblem. It can be used in a user-written callback during MIP optimization. The indicators are from the original problem if `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF`. Otherwise, they are from the presolved problem.

This routine may be called only when the value of the `wherefrom` argument is one of `CPX_CALLBACK_MIP`, `CPX_CALLBACK_MIP_BRANCH`, `CPX_CALLBACK_MIP_INCUMBENT`, `CPX_CALLBACK_MIP_NODE`, `CPX_CALLBACK_MIP_HEURISTIC`, or `CPX_CALLBACK_MIP_CUT`.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetcallbacknodeintfeas (CPXCENVptr env,
                               void *cbdata,
                               int wherefrom,
                               int *feas,
                               int begin,
                               int end);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter must be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating from where the user-written callback was called. The parameter must be the value of `wherefrom` passed to the user-written callback.

`int *feas`

An array to receive an indicator of feasibility for the node subproblem. This array must be of length at least (`end - begin + 1`). If successful, `feas[0]` through `feas[end-begin]` will contain the indicators.

Values for `feas[j]`:

`CPX_INTEGER_FEASIBLE`	0	variable j+begin is integer-valued
`CPX_INTEGER_INFEASIBLE`	1	variable j+begin is not integer-valued

`CPX_IMPLIED_INTEGER_FEASIBLE`	2	variable j+begin may have a fractional value in the current solution, but it will take on an integer value when all integer variables still in the problem have integer values. It should not be branched upon.

`int begin`

An integer indicating the beginning of the range of feasibility indicators to be returned.

`int end`

An integer indicating the end of the range of feasibility indicators to be returned.

Example

```
status = CPXgetcallbacknodeintfeas(env, cbdata, wherefrom,
                                   feas, 0, cols-1);
```

See Also

Examples `admipex1.c` *and* `admipex2.c` *in the advanced examples directory*

CPXgetcallbacknodelb

Usage Mixed Integer Users Only

Description The routine `CPXgetcallbacknodelb()` is used to get the lower bound values for the subproblem at the current node during MIP optimization from within a user-written callback. The values are from the original problem if `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF`, otherwise they are from the presolved problem.

This routine may be called only when the value of the `wherefrom` argument is one of `CPX_CALLBACK_MIP`, `CPX_CALLBACK_MIP_BRANCH`, `CPX_CALLBACK_MIP_INCUMBENT`, `CPX_CALLBACK_MIP_NODE`, `CPX_CALLBACK_MIP_HEURISTIC`, `CPX_CALLBACK_MIP_SOLVE`, or `CPX_CALLBACK_MIP_CUT`.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetcallbacknodelb (CPXENVptr env,
                          void *cbdata,
                          int wherefrom,
                          double *lb,
                          int begin,
                          int end);
```

Arguments

`CPXENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter must be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating from where the user-written callback was called. The parameter must be the value of `wherefrom` passed to the user-written callback.

`double *lb`

An array to receive the values of the lower bound values. This array must be of length at least (`end - begin + 1`). If successful, `lb[0]` through `lb[end-begin]` contain the lower bound values for the current subproblem.

`int begin`

An integer indicating the beginning of the range of lower bounds to be returned.

`int end`

An integer indicating the end of the range of lower bounds to be returned.

Example

```
status = CPXgetcallbacknodelb (env, cbdata, wherefrom,
                               lb, 0, cols-1);
```

CPXgetcallbacknodelp

Usage

Mixed Integer Users Only

Description

The routine `CPXgetcallbacknodelp()` accesses the `lp` pointer indicating the currently defined linear programming subproblem (LP) from within user-written callbacks. Generally, this pointer may be used only in ILOG CPLEX Callable Library query routines, such as `CPXsolution()` or `CPXgetrows()`, documented in the *ILOG CPLEX Reference Manual*.

`CPXgetcallbacknodelp()` may be called only when its `wherefrom` argument has one of the following values:

`CPX_CALLBACK_MIP_BRANCH`

`CPX_CALLBACK_MIP_CUT`

`CPX_CALLBACK_MIP_HEURISTIC`

`CPX_CALLBACK_MIP_SOLVE`

When the `wherefrom` argument has the value `CPX_CALLBACK_MIP_SOLVE`, the subproblem pointer may also be used in ILOG CPLEX optimization routines.

> ***Warning:*** *Any modification to the subproblem may result in corruption of the problem and of the ILOG CPLEX environment.*

Return Value

The routine returns a zero on success, and a nonzero if an error occurs. A nonzero return value may mean that the requested value is not available.

Synopsis

```
int CPXgetcallbacknodelp (CPXCENVptr env,
                          void *cbdata,
                          int wherefrom,
                          CPXLPptr *curlp_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The `cbdata` pointer passed to the user-written callback. This parameter *must* be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating where the user-written callback was called from. This parameter *must* be the value of the wherefrom passed to the user-written callback.

`CPXLPptr *curlp_p`

The `lp` pointer indicating the current subproblem. If no subproblem is defined, the pointer is set to `NULL`.

Example

```
status = CPXgetcallbacknodelp (env, cbdata, &curlp);
```

See Also

The examples `admipex1.c` *and* `admipex6.c`.

CPXsetbranchcallbackfunc(), CPXsetcutcallbackfunc(), CPXsetheuristiccallbackfunc(), CPXsetsolvecallbackfunc(), Advanced MIP Control Interface

CPXgetcallbacknodeobjval

Usage Mixed Integer Users Only

Description The routine `CPXgetcallbacknodeobjval()` is used to get the objective value for the subproblem at the current node during MIP optimization from within a user-written callback.

This routine may be called only when the value of the `wherefrom` argument is one of `CPX_CALLBACK_MIP`, `CPX_CALLBACK_MIP_BRANCH`, `CPX_CALLBACK_MIP_INCUMBENT`, `CPX_CALLBACK_MIP_NODE`, `CPX_CALLBACK_MIP_HEURISTIC`, or `CPX_CALLBACK_MIP_CUT`.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetcallbacknodeobjval (CPXCENVptr env,
                              void *cbdata,
                              int wherefrom,
                              double *objval_p);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter must be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating from where the user-written callback was called. The parameter must be the value of `wherefrom` passed to the user-written callback.

`double *objval_p`

A pointer to a variable of type double where the objective value of the node subproblem is to be stored.

Example

```
status = CPXgetcallbacknodeobjval (env, cbdata, wherefrom,
                                   &objval);
```

See Also *Examples* `admipex1.c` *and* `admipex3.c` *in the advanced examples directory*

CPXgetcallbacknodestat

Usage Mixed Integer Users Only

Description The routine `CPXgetcallbacknodestat()` is used to get the optimization status of the subproblem at the current node from within a user-written callback during MIP optimization.

The optimization status will be either optimal or unbounded. An unbounded status can occur when some of the constraints are being treated as lazy constraints. When the node status is unbounded, then the function `CPXgetcallbacknodex()` returns a ray that can be used to decide which lazy constraints need to be added to the subproblem.

This routine may be called only when the value of the `wherefrom` argument is `CPX_CALLBACK_MIP_CUT`.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetcallbacknodestat (CPXCENVptr env,
                            void *cbdata,
                            int wherefrom,
                            int *nodestat_p);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter must be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating from where the user-written callback was called. The parameter must be the value of `wherefrom` passed to the user-written callback.

`int *nodestat_p`

A pointer to an integer where the node subproblem optimization status is to be returned. The values of `*nodestat_p` may be `CPX_OPTIMAL` or `CPX_UNBOUNDED`.

Example

```
status = CPXgetcallbacknodestat (env, cbdata, wherefrom,
                                 &nodestatus);
```

CPXgetcallbacknodeub

Usage

Mixed Integer Users Only

Description

The routine `CPXgetcallbacknodeub()` is used to get the upper bound values for the subproblem at the current node during MIP optimization from within a user-written callback. The values are from the original problem if `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF`, otherwise they are from the presolved problem.

This routine may be called only when the value of the `wherefrom` argument is one of `CPX_CALLBACK_MIP`, `CPX_CALLBACK_MIP_BRANCH`, `CPX_CALLBACK_MIP_INCUMBENT`, `CPX_CALLBACK_MIP_NODE`, `CPX_CALLBACK_MIP_HEURISTIC`, `CPX_CALLBACK_MIP_SOLVE`, or `CPX_CALLBACK_MIP_CUT`.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetcallbacknodeub (CPXCENVptr env,
                          void *cbdata,
                          int wherefrom,
                          double *ub,
                          int begin,
                          int end);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter must be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating from where the user-written callback was called. The parameter must be the value of `wherefrom` passed to the user-written callback.

`double *ub`

An array to receive the values of the upper bound values. This array must be of length at least (`end - begin + 1`). If successful, `ub[0]` through `ub[end-begin]` contain the upper bound values for the current subproblem.

`int begin`

An integer indicating the beginning of the range of upper bound values to be returned.

`int end`

An integer indicating the end of the range of upper bound values to be returned.

Example

```
status = CPXgetcallbacknodeub (env, cbdata, wherefrom,
                               ub, 0, cols-1);
```

CPXgetcallbacknodex

Usage

Mixed Integer Users Only

Description

The routine `CPXgetcallbacknodex()` is used to get the primal variable (x) values for the subproblem at the current node during MIP optimization from within a user-written callback. The values are from the original problem if `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF`, otherwise they are from the presolved problem.

This routine may be called only when the value of the `wherefrom` argument is one of `CPX_CALLBACK_MIP`, `CPX_CALLBACK_MIP_BRANCH`, `CPX_CALLBACK_MIP_INCUMBENT`, `CPX_CALLBACK_MIP_NODE`, `CPX_CALLBACK_MIP_HEURISTIC`, or `CPX_CALLBACK_MIP_CUT`.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetcallbacknodex (CPXCENVptr env,
                         void *cbdata,
                         int wherefrom,
                         double *x,
                         int begin,
                         int end);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter must be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating from where the user-written callback was called. The parameter must be the value of `wherefrom` passed to the user-written callback.

`double *x`

An array to receive the values of the primal variables for the node subproblem. This array must be of length at least (`end - begin + 1`). If successful, `x[0]` through `x[end-begin]` contain the primal values.

`int begin`

An integer indicating the beginning of the range of primal variable values for the node subproblem to be returned.

`int end`

An integer indicating the end of the range of primal variable values for the node subproblem to be returned.

Example

```
status = CPXgetcallbacknodex (env, cbdata, wherefrom,
                              nodex, 0, cols-1);
```

See Also

Examples `admipex1.c`, `admipex3.c`, *and* `admipex5.c` *in the advanced examples directory*

CPXgetcallbackorder

Usage

Mixed Integer Users Only

Description

The routine `CPXgetcallbackorder()` is used to get MIP priority order information during MIP optimization from within a user-written callback. The values are from the original problem if `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF`, otherwise they are from the presolved problem.

This routine may be called only when the value of the `wherefrom` argument is one of `CPX_CALLBACK_MIP`, `CPX_CALLBACK_MIP_BRANCH`, `CPX_CALLBACK_MIP_INCUMBENT`, `CPX_CALLBACK_MIP_NODE`, `CPX_CALLBACK_MIP_HEURISTIC`, `CPX_CALLBACK_MIP_SOLVE`, or `CPX_CALLBACK_MIP_CUT`.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetcallbackorder (CPXCENVptr env,
                         void *cbdata,
                         int wherefrom,
                         int *priority,
                         int *direction,
                         int begin,
                         int end);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter must be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating from where the user-written callback was called. The parameter must be the value of `wherefrom` passed to the user-written callback.

`int *priority`

An array where the priority values are to be returned. This array must be of length at least (`end - begin + 1`). If successful, `priority[0]` through `priority[end-begin]` contain the priority order values. May be `NULL`.

`int *direction`

An array where the preferred direction values are to be returned. This array must be of length at least (`end - begin + 1`). If successful, `direction[0]` through `direction[end-begin]` contain the preferred direction values. May be `NULL`. The value of direction[j] will be

`CPX_BRANCH_GLOBAL`	0	use global branching direction setting `CPX_PARAM_BRDIR`
`CPX_BRANCH_DOWN`	-1	branch down first on variable `j+begin`
`CPX_BRANCH_UP`	1	branch up first on variable `j+begin`

`int begin`

An integer indicating the beginning of the range of priority order information to be returned.

`int end`

An integer indicating the end of the range of priority order information to be returned.

Example

```
status = CPXgetcallbackorder (env, cbdata, wherefrom,
                              priority, NULL, 0, cols-1);
```

CPXgetcallbackpseudocosts

Usage

Mixed Integer Users Only

Description

The routine `CPXgetcallbackpseudocosts()` is used to get the pseudo-cost values during MIP optimization from within a user-written callback. The values are from the original problem if `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF`. Otherwise, they are from the presolved problem.

> ***Note:*** *When pseudo-costs are retrieved for the original problem variables, pseudo-costs are zero for variables that have been removed from the problem, since they are never used for branching.*

This routine may be called only when the value of the `wherefrom` argument is one of `CPX_CALLBACK_MIP`, `CPX_CALLBACK_MIP_BRANCH`, `CPX_CALLBACK_MIP_INCUMBENT`, `CPX_CALLBACK_MIP_NODE`, `CPX_CALLBACK_MIP_HEURISTIC`, `CPX_CALLBACK_MIP_SOLVE`, or `CPX_CALLBACK_MIP_CUT`.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetcallbackpseudocosts (CPXCENVptr env,
                               void *cbdata,
                               int wherefrom,
                               double *uppc,
                               double *downpc,
                               int begin,
                               int end);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter must be the value of cbdata passed to the user-written callback.

`int wherefrom`

An integer value indicating from where the user-written callback was called. The parameter must be the value of wherefrom passed to the user-written callback.

`double *uppc`

An array to receive the values of up pseudo-costs. This array must be of length at least (`end - begin + 1`). If successful, `uppc[0]` through `uppc[end-begin]` will contain the up pseudo-costs. May be `NULL`.

`double *downpc`

An array to receive the values of the down pseudo-costs. This array must be of length at least (`end - begin + 1`). If successful, `downpc[0]` through `downpc[end-begin]` will contain the down pseudo-costs. May be `NULL`.

`int begin`

An integer indicating the beginning of the range of pseudo-costs to be returned.

`int end`

An integer indicating the end of the range of pseudo-costs to be returned.

Example

```
status = CPXgetcallbackpseudocosts (env, cbdata, wherefrom,
                                    upcost, downcost,
                                    j, k);
```

CPXgetcallbackseqinfo

Usage Mixed Integer Users Only

Description The routine `CPXgetcallbackseqinfo()` accesses information about nodes during the MIP optimization from within user-written callbacks. This routine may be called only when the value of its wherefrom argument is `CPX_CALLBACK_MIP_NODE`. The information accessed from this routine can also be accessed with the routine `CPXgetcallbacknodeinfo()`. Nodes are not stored by sequence number but by node number, so using the routine `CPXgetcallbackseqinfo()` can be much more time-consuming than using the routine `CPXgetcallbacknodeinfo()`. A typical use of this routine would be to obtain the node number of a node for which the sequence number is known and then use that node number to select the node with the node callback.

> ***Note:*** *This routine cannot retrieve information about nodes that have been moved to node files. For more information about node files, see the CPLEX User's Manual. If the argument* `seqnum` *refers to a node in a node file,* `CPXgetcallbacknodeinfo()` *returns the value* `CPXERR_NODE_ON_DISK`.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

The return value `CPXERR_NODE_ON_DISK` indicates an attempt to access a node currently located in a node file on disk.

Synopsis

```
int CPXgetcallbackseqinfo (CPXCENVptr env,
                           void *cbdata,
                           int wherefrom,
                           int seqnum,
                           int whichinfo,
                           void *result_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter *must* be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating where the user-written callback was called from. This parameter *must* be the value of `wherefrom` passed to the user-written callback.

`int seqnum`

The sequence number of the node for which information is requested.

`int whichinfo`

An integer indicating which information is requested. For a summary of possible values, refer to the table *Information Requested for a User-Written Node Callback* in the description of `CPXgetcallbacknodeinfo()`.

`void *result_p`

A generic pointer to a variable of type `double` or `int`, representing the value returned by `whichinfo`. The column *C Type* in the table *Information Requested for a User-Written Node Callback* indicates the type of various values returned by `whichinfo`.

CPXgetcallbacksosinfo

Usage Mixed Integer Users Only

Description The routine `CPXgetcallbacksosinfo()` accesses information about special ordered sets (SOSs) during MIP optimization from within user-written callbacks. This routine may be called only when the value of its `wherefrom` argument is one of these values: `CPX_CALLBACK_MIP_HEURISTIC`, `CPX_CALLBACK_MIP_BRANCH`, or `CPX_CALLBACK_MIP_CUT`.

The information returned is for the original problem if the parameter `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF` before the call to `CPXmipopt()` that calls the callback. Otherwise, it is for the presolved problem.

Return Value The routine returns a zero on success, and a nonzero if an error occurs. If the return value is nonzero, the requested value may not be available.

Synopsis

```
int CPXgetcallbacksosinfo (CPXCENVptr env,
                           void *cbdata,
                           int wherefrom,
                           int sosindex,
                           int member,
                           int whichinfo,
                           void *result_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

The pointer passed to the user-written callback. This parameter *must* be the value of `cbdata` passed to the user-written callback.

`int wherefrom`

An integer value indicating where the user-written callback was called from. This parameter *must* be the value of `wherefrom` passed to the user-written callback.

`int sosindex`

The index of the special ordered set (SOS) for which information is requested. SOSs are indexed from `0` (zero) to `(numsets - 1)` where `numsets` is the result of calling this routine with a `whichinfo` value of `CPX_CALLBACK_INFO_SOS_NUM`.

`int member`

The index of the member of the SOS for which information is requested.

`int whichinfo`

An integer indicating which information is requested. Table 3 summarizes possible values.

Table 3 *Information Requested for a User-Written SOS Callback*

Symbolic Constant	C Type	Meaning
`CPX_CALLBACK_INFO_SOS_NUM`	`int`	number of SOSs
`CPX_CALLBACK_INFO_SOS_TYPE`	`char`	one of the values in Table 4
`CPX_CALLBACK_INFO_SOS_SIZE`	`int`	size of SOS
`CPX_CALLBACK_INFO_SOS_IS_FEASIBLE`	`int`	1 if SOS is feasible 0 if SOS is not
`CPX_CALLBACK_INFO_SOS_PRIORITY`	`int`	priority value of SOS
`CPX_CALLBACK_INFO_SOS_MEMBER_INDEX`	`int`	variable index of *member*th member of SOS
`CPX_CALLBACK_INFO_SOS_MEMBER_REFVAL`	`double`	reference value (weight) of this member

Table 4 summarizes possible values returned when the type of information requested is the SOS type (that is, `whichinfo` = `CPX_CALLBACK_INFO_SOS_TYPE`).

Table 4 *SOS Types Returned when* `whichinfo = CPX_CALLBACK_INFO_SOS_TYPE`

Symbolic Constant	SOS Type
`CPX_SOS1`	type 1
`CPX_SOS2`	type 2

`void *result_p`

A generic pointer to a variable of type `double`, `int`, or `char` representing the value returned by `whichinfo`. (The column “C Type” in Table 3 indicates the type of various values returned by `whichinfo`.)

Example

```
status = CPXgetcallbacksosinfo(env, curlp, wherefrom, 6, 4,
                               CPX_CALLBACK_INFO_SOS_IS_FEASIBLE,
                               &isfeasible);
```

See Also

The example `admipex3.c`

Advanced MIP Control Interface

CPXgetcutcallbackfunc

Usage

Mixed Integer Users Only

Description

The routine `CPXgetcutcallbackfunc()` accesses the user-written callback for adding cuts. The user-written callback is called by ILOG CPLEX during MIP branch & cut for every node that has an LP optimal solution with objective value below the cutoff and that is integer infeasible. The callback routine adds globally valid cuts to the LP subproblem.

For documentation of callback arguments, see the routine `CPXsetcutcallbackfunc()`.

Return Value

This routine does not return a result.

Synopsis

```
void CPXgetcutcallbackfunc(CPXCENVptr env,
                           int (**callback_p)
                                 (CPXCENVptr xenv,
                                  void *cbdata,
                                  int wherefrom,
                                  void *cbhandle,
                                  int *useraction_p),
                           void **cbhandle_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`int (**callback_p)`

The address of the pointer to the current user-written cut callback. If no callback has been set, the pointer evaluates to `NULL`.

`void **cbhandle_p`

The address of a variable to hold the user's private pointer.

Example

```
CPXgetcutcallbackfunc(env, &current_cutfunc, &current_data);
```

See Also

CPXcutcallbackadd(), CPXgetcallbacksosinfo(), CPXgetcallbacknodelp(), CPXsetcutcallbackfunc(), Advanced MIP Control Interface

CPXgetdeletenodecallbackfunc

Usage

Mixed Integer Users Only

Description

The routine `CPXgetdeletenodecallbackfunc()` accesses the user-written callback routine to be called during MIP optimization when a node is to be deleted. Nodes are deleted when a branch is carried out from that node, when the node relaxation is infeasible, or when the node relaxation objective value is worse than the cutoff. This callback can be used to delete user data associated with a node.

For documentation of callback arguments, see the routine `CPXsetdeletenodecallbackfunc()`.

Return Value

This routine does not return a result.

Synopsis

```
void CPXgetdeletenodecallbackfunc(CPXCENVptr env,
                                  int (CPXPUBLIC
                                       **deletenodecallback_p)
                                         (CPXCENVptr env,
                                          void *cbdata,
                                          int wherefrom,
                                          void *cbhandle,
                                          int seqnum,
                                          void *handle),
                                  void **cbhandle_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`int (CPXPUBLIC **deletenodecallback_p)`

The address of the pointer to the current user-written delete node callback. If no callback has been set, the pointer evaluates to `NULL`.

`void **cbhandle_p`

The address of a variable to hold the user's private pointer.

Example

```
CPXgetdeletenodecallbackfunc(env,
                             &current_callback,
                             &current_handle);
```

See Also

CPXsetdeletenodecallbackfunc(), CPXbranchcallbackbranchbds(), CPXbranchcallbackbranchconstraints(), CPXbranchcallbackbranchgeneral(), Advanced MIP Control Interface

CPXgetdnorms

Usage

Advanced

Description

The routine `CPXgetdnorms()` accesses the norms from the dual steepest edge. As in `CPXcopydnorms()`, the argument `head` is an array of column or row indices corresponding to the array of norms. Column indices are indexed with nonnegative values. Row indices are indexed with negative values offset by 1 (one). For example, if `head[0] = -5`, `norm[0]` is associated with `row 4`.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetdnorms (CPXCENVptr env,
                  CPXCLPptr lp,
                  double *norm,
                  int *head,
                  int *len_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to the CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`double *norm`

An array containing the dual steepest-edge norms in the ordered specified by `head[]`. The array must be of length at least equal to the number of rows in the LP problem object.

`int *head`

An array containing column or row indices. The allocated length of the array must be at least equal to the number of rows in the LP problem object.

`int *len_p`

A pointer to an integer that indicates the number of entries in both `norm[]` and `head[]`. The value assigned to the pointer `*len_p is` needed by the routine `CPXcopydnorms()`.

See Also

CPXcopydnorms()

CPXgetExactkappa

Usage Advanced

Description The routine `CPXgetExactkappa()` computes and returns the condition number, kappa.

Return Value If successful, this routine returns the condition number, kappa.

Synopsis

```
double CPXgetExactkappa (CPXCENVptr env,
                         CPXCLPptr lp);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

See Also *CPXgetkappa()*

CPXgetheuristiccallbackfunc

Usage

Mixed Integer Users Only

Description

The routine `CPXgetheuristiccallbackfunc()` accesses the user-written callback to be called by ILOG CPLEX during MIP optimization after the subproblem has been solved to optimality. That callback is *not* called when the subproblem is infeasible or cut off. The callback supplies ILOG CPLEX with heuristically-derived integer solutions.

For documentation of callback arguments, see the routine `CPXsetheuristiccallbackfunc()`.

Return Value

This routine does not return a result.

Synopsis

```
void CPXgetheuristiccallbackfunc(CPXCENVptr env,
                                 int (CPXPUBLIC **heuristiccallback_p)
                                        (CPXCENVptr env,
                                         void *cbdata,
                                         int wherefrom,
                                         void *cbhandle,
                                         double *objval_p,
                                         double *x,
                                         int *checkfeas_p,
                                         int *useraction_p),
                                  void **cbhandle_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`int (CPXPUBLIC **heuristiccallback_p)`

The address of the pointer to the current user-written heuristic callback. If no callback has been set, the pointer evaluates to `NULL`.

`void **cbhandle_p`

The address of a variable to hold the user's private pointer.

Example

```
CPXgetheuristiccallbackfunc(env, &current_callback,
&current_handle);
```

See Also

CPXsetheuristiccallbackfunc(), Advanced MIP Control Interface

CPXgetijdiv

Usage Advanced

Description The routine `CPXgetijdiv()` returns the index of the diverging row (that is, constraint) or column (that is, variable) when one of the ILOG CPLEX simplex optimizers terminates due to a diverging vector. This function can be called after an unbounded solution status for a primal simplex call or after an infeasible solution status for a dual simplex call.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetijdiv (CPXENVptr env,
                 CPXLPptr lp,
                 int *idiv_p,
                 int *jdiv_p);
```

Arguments

`CPXENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to the CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int *idiv_p`

A pointer to an integer indexing the row of a diverging variable. If one of the ILOG CPLEX simplex optimizers has concluded that the LP problem object is unbounded, and if the diverging variable is a slack or ranged variable, `CPXgetijdiv()` returns the index of the corresponding row in `*idiv_p`. Otherwise, `*idiv_p` is set to `-1`.

`int *jdiv_p`

A pointer to an integer indexing the row of a diverging variable. If one of the ILOG CPLEX simplex optimizers has concluded that the LP problem object is unbounded, and if the diverging variable is a normal, structural variable, `CPXgetijdiv()` sets `*jdiv_p` to the index of that variable. Otherwise, `*jdiv_p` is set to `-1`.

CPXgetijrow

Usage Advanced

Description The routine `CPXgetijrow()` returns the index of a specific basic variable as its position in the basis header. If the specified row indexes a constraint that is not basic, or if the specified column indexes a variable that is not basic, `CPXgetijrow()` returns an error code and sets the value of its argument `*row_p` to `-1`. An error is also returned if both row and column indices are specified in the same call.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetijrow (CPXCENVptr env,
                 CPXCLPptr lp,
                 int i,
                 int j,
                 int *row_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

The pointer to the CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int i`

An integer specifying the index of a basic row; `CPXgetijrow()` must find the position of this basic row in the basis header. A negative value in this argument indicates to `CPXgetijrow()` not to seek a basic row.

`int j`

An integer specifying the index of a basic column; `CPXgetijrow()` must find the position of this basic column in the basis header. A negative value in this argument indicates to `CPXgetijrow()` not to seek a basic column.

`int *row_p`

A pointer to an integer indicating the position in the basis header of the row `i` or column `j`. If `CPXgetijrow()` encounters an error, and if `row_p` is not `NULL`, `*row_p` is set to `-1`.

CPXgetincumbentcallbackfunc

Usage Mixed Integer Users Only

Description The routine `CPXgetincumbentcallbackfunc()` accesses the user-written callback to be called by CPLEX during MIP optimization after an integer solution has been found but before this solution replaces the incumbent. This callback can be used to discard solutions that do not meet criteria beyond that of the mixed integer programming formulation.

For documentation of callback arguments, see the routine `CPXsetincumbentcallbackfunc()`.

Return Value This routine does not return a result.

Synopsis

```
void CPXgetincumbentcallbackfunc(CPXCENVptr env,
                                 int (CPXPUBLIC **incumbentcallback_p)
                                         (CPXCENVptr xenv,
                                          void *cbdata,
                                          int wherefrom,
                                          void *cbhandle,
                                          double objval,
                                          double *x,
                                          int *isfeas_p,
                                          int *useraction_p),
                                 void **cbhandle_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`int (CPXPUBLIC **incumbentcallback_p)`

The address of the pointer to the current user-written incumbent callback. If no callback has been set, the pointer evaluates to `NULL`.

`void **cbhandle_p`

The address of a variable to hold the user's private pointer.

Example

```
CPXgetincumbentcallbackfunc(env, &current_incumbentcallback,
&current_handle);
```

See Also *CPXsetincumbentcallbackfunc(), Advanced MIP Control Interface*

CPXgetkappa

Usage Advanced

Description The routine `CPXgetkappa()` computes and returns an estimate of the condition number, kappa.

Return Value If successful, this routine returns an estimate of the condition number, kappa.

Synopsis

```
double CPXgetkappa (CPXCENVptr env,
                    CPXCLPptr lp);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to the CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

CPXgetnodecallbackfunc

Usage Mixed Integer Users Only

Description The routine `CPXgetnodecallbackfunc()` accesses the user-written callback to be called during MIP optimization after ILOG CPLEX has selected a node to explore, but before this exploration is carried out. The callback routine can change the node selected by ILOG CPLEX to a node selected by the user.

For documentation of callback arguments, see the routine `CPXsetnodecallbackfunc()`.

Return Value This routine does not return a result.

Synopsis

```
void CPXgetnodecallbackfunc(CPXCENVptr env,
                            int (CPXPUBLIC **nodecallback_p)
                                  (CPXCENVptr env,
                                   void *cbdata,
                                   int wherefrom,
                                   void *cbhandle,
                                   int *nodeindex_p,
                                   int *useraction_p),
                            void **cbhandle_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`int (CPXPUBLIC **nodecallback_p)`

The address of the pointer to the current user-written node callback. If no callback has been set, the pointer will evaluate to `NULL`.

`void **cbhandle_p`

The address of a variable to hold the user's private pointer.

Example

```
CPXgetnodecallbackfunc(env, &current_callback, &current_handle);
```

See Also *The example admipex1.c.*

CPXsetnodecallbackfunc(), CPXgetcallbacknodeinfo(), Advanced MIP Control Interface

CPXgetobjoffset

Usage Advanced

Description The routine `CPXgetobjoffset()` returns the objective offset between the original problem and the presolved problem.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetobjoffset (CPXCENVptr env,
                     CPXCLPptr redlp,
                     double *objoffset_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr redlp`

A pointer to a reduced CPLEX LP problem object, as returned by `CPXgetredlp()`.

`double *objoffset_p`

A pointer to a variable of type `double` to hold the objective offset value.

CPXgetpnorms

Usage Advanced

Description The routine `CPXgetpnorms()` returns the norms from the primal steepest-edge.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetpnorms (CPXCENVptr env,
                  CPXCLPptr lp,
                  double *cnorm,
                  double *rnorm,
                  int *len_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`double *cnorm`

An array containing the primal steepest-edge norms for the normal, column variables. The array must be of length at least equal to the number of columns in the LP problem object.

`double *rnorm`

An array containing the primal steepest-edge norms for ranged variables and slacks. The array must be of length at least equal to the number of rows in the LP problem object.

`int *len_p`

A pointer to the number of entries in the array `cnorm[]`. When this routine is called, `*len_p` is equal to the number of columns in the LP problem object when optimization occurred. The routine `CPXcopypnorms()` needs the value `*len_p`.

There is no comparable argument in this routine for `rnorm[]`. If the rows of the problem have changed since the norms were computed, they are generally no longer valid. However, if columns have been deleted, or if columns have been added, the norms for all remaining columns present before the deletions or additions remain valid.

See Also *CPXcopypnorms()*

CPXgetprestat

Usage Advanced

Description The routine `CPXgetprestat()` is used to get presolve status information for the columns and rows of the presolved problem in the original problem and of the original problem in the presolved problem.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetprestat (CPXCENVptr env,
                   CPXCLPptr lp,
                   int *prestat_p,
                   int *pcstat,
                   int *prstat,
                   int *ocstat,
                   int *orstat);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to the original CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int *prestat_p`

A pointer to an integer that will receive the status of the presolved problem associated with LP problem object lp. May be NULL.

The values for `*prestat_p` are:

0	lp is not presolved or there were no reductions
1	lp has a presolved problem
2	lp was reduced to an empty problem

`int *pcstat`

The array where the presolve statuses of the columns are to be returned. The array must be of length at least the number of columns in the original problem object. May be NULL.

For variable `i` in the original problem, values for `pcstat[i]`:

	≥0	variable `i` corresponds to variable `pcstat[i]` in the presolved problem
`CPX_PRECOL_LOW`	-1	variable `i` is fixed to its lower bound
`CPX_PRECOL_UP`	-2	variable `i` is fixed to its upper bound
`CPX_PRECOL_FIX`	-3	variable `i` is fixed to some other value
`CPX_PRECOL_AGG`	-4	variable `i` is aggregated out
`CPX_PRECOL_OTHER`	-5	variable `i` is deleted or merged for some other reason

`int *prstat`

The array where the presolve statuses of the rows are to be returned. The array must be of length at least the number of rows in the original problem object. May be NULL.

For row i in the original problem, values for `prstat[i]`:

	≥0	row `i` corresponds to row `prstat[i]` in the original problem
`CPX_PREROW_RED`	-1	if row `i` is redundant
`CPX_PREROW_AGG`	-2	if row `i` is used for aggregation
`CPX_PREROW_OTHER`	-3	if row `i` is deleted for some other reason

`int *ocstat`

The array where the presolve statuses of the columns of the presolved problem are to be returned. The array must be of length at least the number of columns in the presolved problem object. May be NULL.

For variable i in the presolved problem, values for `ocstat[i]`:

≥0	variable `i` in the presolved problem corresponds to variable `ocstat[i]` in the original problem.
-1	variable `i` corresponds to a linear combination of some variables in the original problem.

`int *orstat`

The array where the presolve statuses of the rows of the presolved problem are to be returned. The array must be of length at least the number of rows in the presolved problem object. May be NULL.

For row i in the original problem, values for `orstat[i]`:

≥0	if row `i` in the presolved problem corresponds to row `orstat[i]` in the original problem
-1	if row `i` is created by, for example, merging two rows in the original problem.

Example

```
status = CPXgetprestat (env, lp, &presolvestat,
                        precstat, prerstat,
                        origcstat, origrstat);
```

See Also

Example `admipex6.c` *in the advanced examples directory*

CPXgetprotected

Usage

Advanced

Description

The routine `CPXgetprotected()` is used to get the set of variables that cannot be aggregated out.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

The value `CPXERR_NEGATIVE_SURPLUS` indicates that insufficient space was available in the array `indices` to hold the protected variable indices.

Synopsis

```
int CPXgetprotected (CPXCENVptr env,
                     CPXCLPptr lp,
                     int *cnt_p,
                     int *indices,
                     int pspace,
                     int *surplus_p);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int *cnt_p`

A pointer to an integer to contain the number of protected variables returned, that is, the true length of the array `indices`.

`int *indices`

The array to contain the indices of the protected variables.

`int pspace`

An integer indicating the length of the array `indices`.

`int *surplus_p`

A pointer to an integer to contain the difference between `pspace` and the number of entries in `indices`. A non-negative value of `*surplus_p` indicates that the length of the arrays was sufficient. A negative value indicates that the length was insufficient and

that the routine could not complete its task. In that case, the routine `CPXgetprotected()` returns the value `CPXERR_NEGATIVE_SURPLUS`, and the value of `*surplus_p` indicates the amount of insufficient space in the arrays.

> ***Note:*** *If the value of* `pspace` *is 0, the negative of the value of* `*surplus_p` *returned indicates the length needed for array* `indices`.

Example

```
status = CPXgetprotected (env, lp, &protectcnt,
                          protectind, 10, &surplus);
```

CPXgetray

Usage Advanced

Description The routine `CPXgetray()` is used to find an unbounded direction or "ray" for a linear program where either the CPLEX primal simplex algorithm concludes that the LP is unbounded (solution status `CPX_UNBOUNDED`). An error is returned, `CPXERR_NOT_UNBOUNDED`, if this case does not hold.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetray (CPXCENVptr env,
               CPXCLPptr lp,
               double *z);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to the CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`double *z`

The array where the unbounded direction is returned. This array must be at least as large as the number of columns in the problem object.

As an illustration, for a linear program of the form

Minimize $c^T x$

Subject to $\boldsymbol{A}x = b$

$x \geq 0$

If the CPLEX primal simplex algorithm completes optimization with a solution status of `CPX_UNBOUNDED`, the vector z returned by `CPXgetray()` would satisfy

$c^T z < 0$

$\boldsymbol{A}z = 0$

$z \geq 0$

if computations could be carried out in exact arithmetic.

Example

```
status = CPXgetray (env, lp, z);
```

CPXgetredlp

Usage

Advanced

Description

The routine `CPXgetredlp()` returns a pointer for the presolved problem. It returns `NULL` if the problem is not presolved or if all the columns and rows are removed by presolve. Generally, the returned pointer may be used only in CPLEX Callable Library query routines, such as `CPXsolution()` or `CPXgetrows()`.

The presolved problem must not be modified. Any modifications must be done on the original problem. If `CPX_PARAM_REDUCE` is set appropriately, the modifications are automatically carried out on the presolved problem at the same time. Optimization and query routines can be used on the presolved problem.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXgetredlp (CPXCENVptr env,
                 CPXCLPptr lp,
                 CPXCLPptr *redlp_p);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`CPXCLPptr *redlp_p`

A pointer to receive the problem object pointer that results when presolve has been applied to the LP problem object.

Example

```
status = CPXgetredlp (env, lp, &reducelp);
```

CPXgetsolvecallbackfunc

Usage

Mixed Integer Users Only

Description

The routine `CPXgetsolvecallbackfunc()` accesses the user-written callback to be called during MIP optimization to optimize the subproblem.

For documentation of callback arguments, see the routine `CPXsetsolvecallbackfunc()`.

Return Value

This routine does not return a result.

Synopsis

```
void CPXgetsolvecallbackfunc(CPXCENVptr env,
                             int (CPXPUBLIC **solvecallback_p)
                                   (CPXCENVptr env,
                                    void *cbdata,
                                    int wherefrom,
                                    void *cbhandle,
                                    int *useraction_p),
                             void **cbhandle_p);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`int (CPXPUBLIC **solvecallback_p)`

The address of the pointer to the current user-written solve callback. If no callback has been set, the pointer evaluates to `NULL`.

`void **cbhandle_p`

The address of a variable to hold the user's private pointer.

Example

```
CPXgetsolvecallbackfunc(env, &current_callback, &current_cbdata);
```

See Also

CPXgetcallbacknodelp(), CPXsetsolvecallbackfunc(), Advanced MIP Control Interface

CPXkilldnorms

Usage Advanced

Description The routine `CPXkilldnorms()` deletes any dual steepest-edge norms that have been retained relative to an active basis. If the user believes that the values of these norms may be significantly in error, and the setting of the `CPX_PARAM_DPRIIND` parameter is `CPX_DPRIIND_STEEP` or `CPX_DPRIIND_FULLSTEEP`, calling `CPXkilldnorms()` means that fresh dual steepest-edge norms will be computed on the next call to `CPXdualopt()`.

Synopsis

```
void  CPXkilldnorms (CPXLPptr lp);
```

Arguments `CPXLPptr lp`

The pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

CPXkillpnorms

Usage Advanced

Example The routine `CPXkillpnorms()` deletes any primal steepest-edge norms that have been retained relative to an active basis. If the user believes that the values of these norms may be significantly in error, and the setting of the `CPX_PARAM_PPRIIND` parameter is `CPX_PPRIIND_STEEP`, calling `CPXkillpnorms()` means that fresh dual steepest-edge norms will be computed on the next call to `CPXprimopt()`.

Synopsis

```
void  CPXkillpnorms (CPXLPptr lp);
```

Arguments `CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

CPXmdleave

Usage

Advanced

Description

The routine `CPXmdleave()` assumes that there is a resident optimal simplex basis, and a resident LU-factorization associated with this basis. It takes as input a list of basic variables as specified by `goodlist[]` and `goodlen`, and returns values commonly known as Driebeek penalties in the two arrays `downratio[]` and `upratio[]`.

For a detailed description of the conditions imposed by this function on `goodlist[]` and `goodlen`, and the detailed meaning of the entries in `downratio[]` and `upratio[]`, see the discussion in the "Arguments" section below.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXmdleave (CPXCENVptr env,
                CPXLPptr lp,
                const int *goodlist,
                int goodlen,
                double *downratio,
                double *upratio);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by`CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`const int *goodlist`

An array of integers that must be of length at least `goodlen`. The entries in `goodlist[]` must all be indices of current basic variables. Moreover, these indices must all be indices of original model variables; that is, they must all take values smaller than the number of columns in the model as returned by `CPXgetnumcols()`. Negative indices and indices bigger than or equal to `CPXgetnumcols()` result in an error.

`int goodlen`

An integer indicating the number of entries in `goodlist[]`. If `goodlen` < 0, an error is returned.

`double *downratio`

An array of type `double` that must be of length at least `goodlen`.

For a given j = goodlist[i], downratio[i] has the following meaning. Let x_j be the name of the basic variable with index *j*, and suppose that x_j is fixed to some value $t' < t$. In a subsequent call to `CPXdualopt()`, the leaving variable in the first iteration of this call is uniquely determined: It must be x_j.

There are then two possibilities. Either an entering variable is determined, or it is concluded (in the first iteration) that the changed model is dual unbounded (primal infeasible). In the latter case, `downratio[i]` is set equal to a large positive value (this number is system dependent, but is usually *1.0E+75*). In the former case, where *r* is the value of the objective function after this one iteration, `downratio[i]` is determined by $|r| = (t - t')$ * `downratio[i]`.

`double *upratio`

An array of type double that must be of length at least `goodlen`. For a given *j* = `goodlist[i]`, `upratio[i]` has the following meaning. Let x_j be the name of the basic variable with index *j*, and suppose that x_j is fixed to some value $t' > t$. Then in a subsequent call to `CPXdualopt()`, the leaving variable in the first iteration of this call is uniquely determined: It must be x_j.

There are then two possibilities. Either an entering variable is determined, or it is concluded (in the first iteration) that the changed model is dual unbounded (primal infeasible). In the latter case, `upratio[i]` is set equal to a large positive value (this number is system dependent, but is usually *1.0E+75*). In the former case, where *r* is the value of the objective function after this one iteration, `upratio[i]` is determined by $|r| = (t' - t)$ * `upratio[i]`.

CPXpivot

Usage

Advanced

Description

The routine `CPXpivot()` performs a basis change where variable `jenter` replaces variable `jleave` in the basis.

It is invalid to pass a basic variable for `jenter`. Also, no nonbasic variable may be specified for `jleave`, except for `jenter == jleave` when the variable has both finite upper and lower bounds. In that case the variable is moved from the current to the other bound. No shifting or perturbation is performed.

Return Value

This routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXpivot (CPXENVptr env,
              CPXLPptr lp,
              int jenter,
              int jleave,
              int leavestat);
```

Arguments

`CPXENVptr env`

The pointer to the CPLEX environment, as returned by the one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int jenter`

An index indicating the variable to enter the basis. The slack or artificial variable for row `i` is denoted by `jenter = -i-1`. `jenter` must identify a nonbasic variable.

`int jleave`

An index indicating the variable to leave the basis. The slack or artificial variable for row `i` is denoted by `jenter = -i-1`. `jleave` must identify a basic variable, unless `jenter` denotes a variable with finite upper and lower bounds. In that case, `jleave` may be set to `jenter` to indicate that the variable is moved from its current bound to the other.

`int leavestat`

An integer indicating the nonbasic status to be assigned to the leaving variable after the basis change. This is important for the case where `jleave` indicates a variable with finite upper and lower bounds, as it may become nonbasic at its lower or upper bound.

Example

```
status = CPXpivot (env, lp, jenter, jleave, CPX_AT_LOWER);
```

CPXpivotin

Usage Advanced

Description The routine `CPXpivotin()` forcibly pivots slacks that appear on a list of inequality rows into the basis. If equality rows appear among those specified on the list, they are ignored.

Motivation

In the implementation of cutting-plane algorithms for integer programming, it is occasionally desirable to delete some of the added constraints (that is, cutting planes) when they no longer appear to be useful. If the slack on some such constraint (that is, row) is not in the resident basis, the deletion of that row may destroy the quality of the basis. Pivoting the slack in before the deletion avoids that difficulty.

Dual Steepest-Edge Norms

If one of the dual steepest-edge algorithms is in use when this routine is called, the corresponding norms are automatically updated as part of the pivot. (Primal steepest-edge norms are not automatically updated in this way because, in general, the deletion of rows invalidates those norms.)

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXpivotin (CPXCENVptr env,
                CPXLPptr lp,
                const int *rlist,
                int rlen);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`const int *rlist`

An array of length `rlen`, containing distinct row indices of slack variables that are not basic in the current solution. If `rlist[]` contains negative entries or entries exceeding the number of rows, `CPXpivotin()` returns an error code. Entries of nonslack rows are ignored.

`int rlen`

An integer that indicates the number of entries in the array `rlist[]`. If `rlen` is negative or greater than the number of rows, `CPXpivotin()` returns an error code.

CPXpivotout

Usage Advanced

Description The routine `CPXpivotout()` pivots a list of fixed variables out of the resident basis. Variables are fixed when the absolute difference between the lower and upper bounds is at most *1.0e-10*.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXpivotout (CPXCENVptr env,
                 CPXLPptr lp,
                 const int *clist,
                 int clen);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`const int *clist`

An array of length `clen`, containing the column indices of the variables to be pivoted out of the basis. If any of these variables is not fixed, `CPXpivotout()` returns an error code.

`int clen`

An integer that indicates the number of entries in the array `clist[]`.

CPXpreaddrows

Usage Advanced

Description The routine `CPXpreaddrows()` is used to add rows to an LP problem object and its associated presolved LP problem object. Note that the CPLEX parameter `CPX_PARAM_REDUCE` must be set to `CPX_PREREDUCE_PRIMALONLY (1)` or `CPX_PREREDUCE_NOPRIMALORDUAL (0)` at the time of the presolve in order to add rows and preserve the presolved problem. This routine should be used in place of `CPXaddrows()` when it is desired to preserve the presolved problem.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXpreaddrows (CPXENVptr env,
                   CPXLPptr lp,
                   int rcnt,
                   int nzcnt,
                   double *rhs,
                   char *sense,
                   int *rmatbeg,
                   int *rmatind,
                   double *rmatval,
                   char **rowname);
```

Arguments The arguments of `CPXpreaddrows()` are the same as those of `CPXaddrows()`, with the exception that new columns may not be added, so there are no `ccnt` and `colname` arguments. The new rows are added to both the original LP problem object and the associated presolved LP problem object.

Examples

```
status = CPXpreaddrows (env, lp, rcnt, nzcnt, rhs, sense,
                        rmatbeg, rmatind, rmatval,
                        newrowname);
```

See Also *Example* `adpreex1.c` *in the advanced examples directory*

CPXprechgobj

Usage Advanced

Description The routine `CPXprechgobj()` is used to change the objective function coefficients of an LP problem object and its associated presolved LP problem object. Note that the CPLEX parameter `CPX_PARAM_REDUCE` must be set to `CPX_PREREDUCE_PRIMALONLY (1)` or `CPX_PREREDUCE_NOPRIMALORDUAL (0)` at the time of the presolve in order to change objective coefficients and preserve the presolved problem. This routine should be used in place of `CPXchgobj()` when it is desired to preserve the presolved problem.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXprechgobj (CPXENVptr env,
                  CPXLPptr lp,
                  int cnt,
                  int *indices,
                  double *values);
```

Arguments The arguments and operation of `CPXprechgobj()` are the same as those of `CPXchgobj()`. The objective coefficient changes are applied to both the original LP problem object and the associated presolved LP problem object.

Example

```
status = CPXprechgobj (env, lp, objcnt, objind, objval);
```

See Also *Example* `adpreex1.c` *in the advanced examples directory*

CPXpresolve

Usage

Advanced

Description

The routine CPXpresolve() performs LP or MIP presolve depending whether a problem object is an LP or a MIP. If the problem is already presolved, the existing presolved problem is freed, and a new presolved problem is created.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXpresolve (CPXCENVptr env,
                 CPXLPptr lp,
                 int method);
```

Arguments

CPXCENVptr env

The pointer to the CPLEX environment, as returned by one of the CPXopenCPLEX routines.

CPXLPptr lp

A pointer to a CPLEX LP problem object, as returned by CPXcreateprob, documented in the *CPLEX Reference Manual*.

int method

An integer specifying the optimization algorithm to be used to solve the problem after the presolve is completed. Some presolve reductions are specific to an optimization algorithm, so specifying the algorithm ensures that the problem is presolved for that algorithm, and that presolve does not have to be re-done when that optimization routine is called. Possible values are CPX_ALG_NONE, CPX_ALG_PRIMAL, CPX_ALG_DUAL, and CPX_ALG_BARRIER for LP; CPX_ALG_NONE should be used for MIP.

Example

```
status = CPXpresolve (env, lp, CPX_ALG_DUAL);
```

CPXqpdjfrompi

Usage

Advanced

Description

The routine `CPXqpdjfrompi()` computes an array of reduced costs from an array of dual values for a QP.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXqpdjfrompi (CPXCENVptr env,
                   CPXCLPptr lp,
                   const double *pi,
                   const double *x,
                   double *dj);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`const double *pi`

An array that contains dual solution (`pi`) values for a problem, as returned by such routines as `CPXqpuncrushpi()` and `CPXcrushpi()`. The length of the array must at least equal the number of rows in the LP problem object.

`const double *x`

An array that contains primal solution (`x`) values for a problem, as returned by such routines as `CPXuncrushx()` and `CPXcrushx()`. The length of the array must at least equal the number of columns in the LP problem object.

`double *dj`

An array to receive the reduced cost values computed from the `pi` values for the problem object. The length of the array must at least equal the number of columns in the problem object.

Example

```
status = CPXqpdjfrompi (env, lp, origpi, reducepi);
```

CPXqpuncrushpi

Usage Advanced

Description The routine `CPXqpuncrushpi()` uncrushes a dual solution for the presolved problem to a dual solution for the original problem if the original problem is a QP.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXqpuncrushpi (CPXENVptr env,
                    CPXLPptr lp,
                    double *pi,
                    const double *prepi
                    const double *x);
```

Arguments

`CPXENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`double *pi`

An array to receive dual solution (`pi`) values for the original problem as computed from the dual values of the presolved problem object. The length of the array must at least equal the number of rows in the LP problem object.

`const double *prepi`

An array that contains dual solution (`pi`) values for the presolved problem, as returned by such routines as `CPXgetpi()` and `CPXsolution()` when applied to the presolved problem object. The length of the array must at least equal the number of rows in the presolved problem object.

`const double *x`

An array that contains primal solution (`x`) values for a problem, as returned by such routines as `CPXuncrushx()` and `CPXcrushx()`. The length of the array must at least equal the number of columns in the LP problem object.

Example

```
status = CPXqpuncrushpi (env, lp, pi, prepi, x);
```

CPXsetbranchcallbackfunc

Usage

Mixed Integer Users Only

Description

The routine `CPXsetbranchcallbackfunc()` sets and modifies the user-written callback routine to be called after a branch has been selected but before the branch is carried out during MIP optimization. In the callback routine, the CPLEX-selected branch can be changed to a user-selected branch.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXsetbranchcallbackfunc (CPXENVptr env,
                              int (CPXPUBLIC *branchcallback)
                                     (CPXCENVptr env,
                                      void *cbdata,
                                      int wherefrom,
                                      void *cbhandle,
                                      int type,
                                      int sos,
                                      int nodecnt,
                                      int bdcnt,
                                      double *nodeest,
                                      int *nodebeg,
                                      int *indices,
                                      char *lu,
                                      int *bd,
                                      int *useraction_p),
                              void *cbhandle);
```

Arguments

`CPXENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`int (CPXPUBLIC *branchcallback)`

A pointer to a user-written branch callback. If the callback is set to `NULL`, no callback can be called during optimization.

`void *cbhandle`

A pointer to user private data. This pointer is passed to the callback.

Callback

Description

The call to the branch callback occurs after a branch has been selected but before the branch is carried out. This function is written by the user. On entry to the callback, the ILOG CPLEX-selected branch is defined in the arguments. The arguments to the callback specify a list of changes to make to the bounds of variables when child nodes are created. One, two, or zero child nodes can be created, so one, two, or zero lists of changes are specified in the arguments. The first branch specified is considered first. The callback is called with zero lists of bound changes when the solution at the node is integer feasible.

Custom branching strategies can be implemented by calling the CPLEX function `CPXbranchcallbackbranchbds()` and setting the `useraction` variable to `CPX_CALLBACK_SET`. Then CPLEX will carry out these branches instead of the CPLEX-selected branches.

Branch variables are in terms of the original problem if the parameter `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF` before the call to `CPXmipopt()` that calls the callback. Otherwise, branch variables are in terms of the presolved problem.

Return Value

The callback returns a zero on success, and a nonzero if an error occurs.

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

A pointer passed from the optimization routine to the user-written callback that identifies the problem being optimized. The only purpose of this pointer is to pass it to the callback information routines.

`int wherefrom`

An integer value indicating where in the optimization this function was called. It will have the value `CPX_CALLBACK_MIP_BRANCH`.

`void *cbhandle`

A pointer to user private data.

`int type`

An integer that indicates the type of branch.Table 5 summarizes possible values.

Table 5 *Branch Types Returned from a User-Written Branch Callback*

Symbolic Constant	Value	Branch
`CPX_TYPE_VAR`	`'0'`	variable branch
`CPX_TYPE_SOS1`	`'1'`	SOS1 branch
`CPX_TYPE_SOS2`	`'2'`	SOS2 branch
`CPX_TYPE_USER`	`'X'`	user-defined

`int sos`

An integer that indicates the special ordered set (SOS) used for this branch. A value of `-1` indicates that this branch is not an SOS-type branch.

`int nodecnt`

An integer that indicates the number of nodes CPLEX will create from this branch. Possible values are `0` (zero), `1`, and `2`. If the argument is `0`, the node will be fathomed unless user-specified branches are made; that is, no child nodes are created and the node itself is discarded.

`int bdcnt`

An integer that indicates the number of bound changes defined in the arrays `indices`, `lu`, and `bd` that define the CPLEX-selected branch.

`double *nodeest`

An array with `nodecnt` entries that contains estimates of the integer objective-function value that will be attained from the created node.

`int *nodebeg`

An array with `nodecnt` entries. The i^{th} entry is the index into the arrays `indices`, `lu`, and `bd` of the first bound changed for the i^{th} node.

`int *indices`

Together with `lu` and `bd`, this array defines the bound changes for each of the created nodes. The entry `indices[i]` is the index for the variable.

`char *lu`

Together with `indices` and `bd`, this array defines the bound changes for each of the created nodes. The entry `lu[i]` is one of the three possible values indicating which bound to change: `L` for lower bound, `U` for upper bound, or `B` for both bounds.

`int *bd`

Together with `indices` and `lu`, this array defines the bound changes for each of the created nodes. The entry `bd[i]` indicates the new value of the bound.

`int *useraction_p`

A pointer to an integer indicating the action for ILOG CPLEX to take at the completion of the user callback. Table 6 summarizes the possible actions.

Table 6 *Actions to be Taken After a User-Written Branch Callback*

Value	Symbolic Constant	Action
0	CPX_CALLBACK_DEFAULT	Use CPLEX-selected branch
1	CPX_CALLBACK_FAIL	Exit optimization
2	CPX_CALLBACK_SET	Use user-selected branch, as defined by calls to `CPXbranchcallbackbranchbds()`
3	CPX_CALLBACK_NO_SPACE	Allocate more space and call callback again

Example

```
status = CPXsetbranchcallbackfunc (env, mybranchfunc, mydata);
```

See Also

The example `admipex1.c`

CPXgetbranchcallbackfunc(), CPXbranchcallbackbranchbds(), Advanced MIP Control Interface

CPXsetcutcallbackfunc

Usage

Mixed Integer Users Only

Description

The routine `CPXsetcutcallbackfunc()` sets and modifies the user-written callback for adding cuts. The user-written callback is called by ILOG CPLEX during MIP branch & cut for every node that has an LP optimal solution with objective value below the cutoff and is integer infeasible. The callback routine adds globally valid cuts to the LP subproblem. The cut may be for the original problem if the parameter `CPX_PARAM_MIPCBREDLP` was set to `CPX_OFF` before the call to `CPXmipopt()` that calls the callback. Otherwise, the cut is for the presolved problem.

Within the user-written cut callback, the routine `CPXgetcallbacknodelp()` and other query routines from the Callable Library access information about the subproblem. The routines `CPXgetcallbacknodeintfeas()` and `CPXgetcallbacksosinfo()` examines the status of integer entities.

The routine `CPXcutcallbackadd()` adds cuts to the problem. Cuts added to the problem are first put into a *cut pool*, so they are not present in the subproblem LP until after the user-written cut callback is finished.

Any cuts that are duplicates of cuts already in the subproblem are not added to the subproblem. Cuts that are added remain part of all subsequent subproblems; there is no cut deletion.

If cuts have been added, the subproblem is re-solved and evaluated, and, if the LP solution is still integer infeasible and not cut off, the cut callback is called again.

If the problem has names, user-added cuts have names of the form `Xnumber` where `number` is a sequence number among all cuts generated.

The parameter `CPX_PARAM_REDUCE` must be set to `CPX_PREREDUCE_PRIMALONLY (1)` or `CPX_PREREDUCE_NOPRIMALORDUAL (0)` if the constraints to be added in the callback are lazy constraints, that is, not implied by the constraints in the constraint matrix. The parameter `CPX_PARAM_PRELINEAR` must be set to 0 if the constraints to be added are in terms of the original problem and the constraints are valid cutting planes.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXsetcutcallbackfunc(CPXENVptr env,
                          int (*callback)
                                 (CPXCENVptr xenv,
                                  void *cbdata,
                                  int wherefrom,
                                  void *cbhandle,
                                  int *useraction_p),
                          void *cbhandle);
```

Arguments

`CPXENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`int (*callback)`

The pointer to the current user-written cut callback. If no callback has been set, the pointer evaluates to `NULL`.

`void *cbhandle`

A pointer to user private data. This pointer is passed to the user-written cut callback.

Callback

Description

ILOG CPLEX calls the cut callback when the LP subproblem for a node has an optimal solution with objective value below the cutoff and is integer infeasible.

Return Value

The callback returns a zero on success, and a nonzero if an error occurs.

Arguments

`CPXCENVptr xenv`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

A pointer passed from the optimization routine to the user-written callback that identifies the problem being optimized. The only purpose of this pointer is to pass it to the callback information routines.

`int wherefrom`

An integer value indicating where in the optimization this function was called. It has the value `CPX_CALLBACK_MIP_CUT`.

`void *cbhandle`

A pointer to user private data.

`int *useraction_p`

A pointer to an integer indicating the action for ILOG CPLEX to take at the completion of the user callback. Table 7 summarizes possible actions.

Table 7 *Actions to be Taken After a User-Written Cut Callback*

Value	Symbolic Constant	Action
`0`	`CPX_CALLBACK_DEFAULT`	Use cuts as added

Table 7 Actions to be Taken After a User-Written Cut Callback

Value	Symbolic Constant	Action
1	CPX_CALLBACK_FAIL	Exit optimization
2	CPX_CALLBACK_SET	Use cuts as added

Example

```
status = CPXsetcutcallbackfunc(env, mycutfunc, mydata);
```

See Also

The example `admipex5.c`

CPXcutcallbackadd(), CPXgetcutcallbackfunc(), Advanced MIP Control Interface

CPXsetdeletenodecallbackfunc

Usage Mixed Integer Users Only

Description The routine `CPXsetdeletenodecallbackfunc()` sets and modifies the user-written callback routine to be called during MIP optimization when a node is to be deleted. Nodes are deleted when a branch is carried out from that node, when the node relaxation is infeasible, or when the node relaxation objective value is worse than the cutoff.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXsetdeletenodecallbackfunc(CPXENVptr env,
                                 int (CPXPUBLIC
                                      *deletenodecallback)
                                         (CPXCENVptr env,
                                          void *cbdata,
                                          int wherefrom,
                                          int seqnum,
                                          void *handle),
                                 void *cbhandle);
```

Arguments

`CPXENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`int (CPXPUBLIC *deletenodecallback)`

A pointer to a user-written branch callback. If the callback is set to `NULL`, no callback can be called during optimization.

`void *cbhandle`

A pointer to user private data. This pointer is passed to the callback.

Callback

Description The call to the delete node callback routine occurs during MIP optimization when a node is to be deleted. Nodes are deleted when a branch is carried out from that node, when the node relaxation is infeasible, or when the node relaxation objective value is worse than the cutoff.

The main purpose of the callback is to provide an opportunity to free any user data associated with the node, thus preventing memory leaks.

Return Value The callback returns a zero on success, and a nonzero if an error occurs.

Arguments `CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

A pointer passed from the optimization routine to the user-written callback that identifies the problem being optimized. The only purpose of this pointer is to pass it to the callback information routines.

`int wherefrom`

An integer value indicating where in the optimization this function was called. It will have the value `CPX_CALLBACK_MIP_DELETENODE`.

`void *cbhandle`

A pointer to user private data.

`int seqnum`

The sequence number of the node that is being deleted.

`void *handle`

A pointer to the user private data that was assigned to the node when it was created with one of the callback branching routines `CPXbranchcallbackbranchbds()`, `CPXbranchcallbackbranchconstraints()`, or `CPXbranchcallbackbranchgeneral()`.

Example

```
status = CPXsetdeletenodecallbackfunc (env,
                                       mybranchfunc,
                                       mydata);
```

See Also *CPXgetdeletenodecallbackfunc(), CPXbranchcallbackbranchbds(), CPXbranchcallbackbranchconstraints(), CPXbranchcallbackbranchgeneral(), Advanced MIP Control Interface*

CPXsetheuristiccallbackfunc

Usage Mixed Integer Users Only

Description The routine `CPXsetheuristiccallbackfunc()` sets or modifies the user-written callback to be called by ILOG CPLEX during MIP optimization after the subproblem has been solved to optimality. That callback is *not* called when the subproblem is infeasible or cut off. The callback supplies ILOG CPLEX with heuristically-derived integer solutions.

If a linear program must be solved as part of a heuristic callback, make a copy of the node LP and solve the copy, not the CPLEX node LP.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXsetheuristiccallbackfunc(CPXENVptr env,
                                int (CPXPUBLIC *heuristiccallback)
                                      (CPXCENVptr env,
                                       void *cbdata,
                                       int wherefrom,
                                       void *cbhandle,
                                       double *objval_p,
                                       double *x,
                                       int *checkfeas_p,
                                       int *useraction_p),
                                void *cbhandle);
```

Arguments

`CPXENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`int (CPXPUBLIC *heuristiccallback)`

A pointer to a user-written heuristic callback. If this callback is set to `NULL`, no callback is called during optimization.

`void *cbhandle`

A pointer to user's private data. This pointer is passed to the callback.

Callback

Description The call to the heuristic callback occurs after an optimal solution to the subproblem has been obtained. The user can provide that solution to start a heuristic for finding an integer solution. The integer solution provided to ILOG CPLEX replaces the incumbent

if it has a better objective value. The basis that is saved as part of the incumbent is the optimal basis from the subproblem; it may not be a good basis for starting optimization of the fixed problem.

The integer solution returned to CPLEX is for the original problem if the parameter `CPX_PARAM_MIPCBREDLP` was set to `CPX_OFF` before the call to `CPXmipopt()` that calls the callback. Otherwise, it is for the presolved problem.

Return Value

The callback returns a zero on success, and a nonzero if an error occurs.

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

A pointer passed from the optimization routine to the user-written callback to identify the problem being optimized. The only purpose of the `cbdata` pointer is to pass it to the callback information routines.

`int wherefrom`

An integer value indicating at which point in the optimization this function was called. It has the value `CPX_CALLBACK_MIP_HEURISTIC` for the heuristic callback.

`void *cbhandle`

A pointer to user private data.

`double *objval_p`

A pointer to a variable that on entry contains the optimal objective value of the subproblem and on return contains the objective value of the integer solution found, if any.

`double *x`

An array that on entry contains primal solution values for the subproblem and on return contains solution values for the integer solution found, if any.

`int *checkfeas_p`

A pointer to an integer that indicates whether or not ILOG CPLEX should check the returned integer solution for integer feasibility. The solution is checked if `checkfeas_p` is nonzero. When the solution is checked and found to be integer infeasible, it is discarded, and optimization continues.

`int *useraction_p`

A pointer to an integer to contain the indicator for the action to be taken on completion of the user callback. Table 8 summarizes possible values.

Table 8 *Actions to be Taken after a User-Written Heuristic Callback*

Value	Symbolic Constant	Action
`0`	`CPX_CALLBACK_DEFAULT`	No solution found
`1`	`CPX_CALLBACK_FAIL`	Exit optimization
`2`	`CPX_CALLBACK_SET`	Use user solution as indicated in return values

Example

```
status = CPXsetheuristiccallbackfunc(env, myheuristicfunc,
mydata);
```

See Also

The example `admipex2.c`

CPXgetheuristiccallbackfunc(), Advanced MIP Control Interface

CPXsetincumbentcallbackfunc

Usage

Mixed Integer Users Only

Description

The routine `CPXsetincumbentcallbackfunc()` sets and modifies the user-written callback routine to be called when an integer solution has been found but before this solution replaces the incumbent. This callback can be used to discard solutions that do not meet criteria beyond that of the mixed integer programming formulation.

Variables are in terms of the original problem if the parameter `CPX_PARAM_MIPCBREDLP` is set to `CPX_OFF` before the call to `CPXmipopt()` that calls the callback. Otherwise, variables are in terms of the presolved problem.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXsetincumbentcallbackfunc (CPXENVptr env,
                                 int (CPXPUBLIC *incumbentcallback)
                                        (CPXCENVptr env,
                                         void *cbdata,
                                         int wherefrom,
                                         void *cbhandle,
                                         double objval,
                                         double *x,
                                         int *isfeas_p,
                                         int *useraction_p),
                                 void *cbhandle);
```

Arguments

`CPXENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`int (CPXPUBLIC *incumbentcallback)`

A pointer to a user-written incumbent callback. If the callback is set to `NULL`, no callback can be called during optimization.

`void *cbhandle`

A pointer to user private data. This pointer is passed to the callback.

Callback

Description

The incumbent callback is called when CPLEX has found an integer solution, but before this solution replaces the incumbent integer solution.

Variables are in terms of the original problem if the parameter CPX_PARAM_MIPCBREDLP is set to CPX_OFF before the call to CPXmipopt() that calls the callback. Otherwise, variables are in terms of the presolved problem.

Return Value

The callback returns a zero on success, and a nonzero if an error occurs.

Arguments

CPXCENVptr env

The pointer to the ILOG CPLEX environment, as returned by one of the CPXopenCPLEX routines.

void *cbdata

A pointer passed from the optimization routine to the user-written callback that identifies the problem being optimized. The only purpose of this pointer is to pass it to the callback information routines.

int wherefrom

An integer value indicating where in the optimization this function was called. It will have the value CPX_CALLBACK_MIP_BRANCH.

void *cbhandle

A pointer to user private data.

double objval

A variable that contains the objective value of the integer solution.

double *x

An array that contains primal solution values for the integer solution.

int *isfeas_p

A pointer to an integer variable that indicates whether or not CPLEX should use the integer solution specified in x to replace the current incumbent. A nonzero value indicates that the incumbent should be replaced by x; a zero value indicates that it should not.

int *useraction_p

A pointer to an integer to contain the indicator for the action to be taken on completion of the user callback. Table 9 summarizes possible values.

Table 9 *Actions to be Taken after a User-Written Incumbent Callback*

Value	Symbolic Constant	Action
0	CPX_CALLBACK_DEFAULT	Proceed with optimization
1	CPX_CALLBACK_FAIL	Exit optimization
2	CPX_CALLBACK_SET	Proceed with optimization

Example

```
status = CPXsetincumbentcallbackfunc (env, myincumbentcheck,
mydata);
```

See Also

CPXgetincumbentcallbackfunc(), Advanced MIP Control Interface

CPXsetnodecallbackfunc

Usage Mixed Integer Users Only

Description The routine `CPXsetnodecallbackfunc()` sets and modifies the user-written callback to be called during MIP optimization after ILOG CPLEX has selected a node to explore, but before this exploration is carried out. The callback routine can change the node selected by ILOG CPLEX to a node selected by the user.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXsetnodecallbackfunc (CPXENVptr env,
                            int (CPXPUBLIC *nodecallback)
                                  (CPXCENVptr env,
                                   void *cbdata,
                                   int wherefrom,
                                   void *cbhandle,
                                   int *nodeindex_p,
                                   int *useraction_p),
                            void *cbhandle);
```

Arguments

`CPXENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`int (CPXPUBLIC *nodecallback)`

A pointer to the current user-written node callback. If no callback has been set, the pointer evaluates to `NULL`.

`void *cbhandle`

A pointer to user private data. This pointer is passed to the user-written node callback.

Callback

Description ILOG CPLEX calls the node callback after selecting the next node to explore. The user can choose another node by setting the argument values of the callback.

Return Value The callback returns a zero on success, and a nonzero if an error occurs.

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

A pointer passed from the optimization routine to the user-written callback that identifies the problem being optimized. The only purpose of this pointer is to pass it to the callback information routines.

`int wherefrom`

An integer value indicating where in the optimization this function was called. It has the value `CPX_CALLBACK_MIP_NODE`.

`void *cbhandle`

A pointer to user private data.

`int *nodeindex_p`

A pointer to an integer that indicates the node number of the user-selected node. The node selected by ILOG CPLEX is node number 0 (zero). Other nodes are numbered relative to their position in the tree, and this number changes with each tree operation. The unchanging identifier for a node is its *sequence number*. To access the sequence number of a node, use the ILOG CPLEX Callable Library routine `CPXgetcallbacknodeinfo()`. An error results if a user attempts to select a node that has been moved to a node file. (See the *CPLEX User's Manual* for more information about node files.)

`int *useraction_p`

A pointer to an integer indicating the action to be taken on completion of the user callback. Table 10 summarizes possible actions.

Table 10 *Actions to be Taken after a User-Written Node Callback*

Value	Symbolic Constant	Action
`0`	`CPX_CALLBACK_DEFAULT`	Use ILOG CPLEX-selected node
`1`	`CPX_CALLBACK_FAIL`	Exit optimization
`2`	`CPX_CALLBACK_SET`	Use user-selected node as defined in returned values

Example

```
status = CPXgetnodecallbackfunc(env, mynodefunc, mydata);
```

See Also

The example `admipex1.c`

CPXgetnodecallbackfunc(), CPXgetcallbacknodeinfo(), Advanced MIP Control Interface

CPXsetsolvecallbackfunc

Usage Mixed Integer Users Only

Description The routine `CPXsetsolvecallbackfunc()` sets and modifies the user-written callback to be called during MIP optimization to optimize the subproblem.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXsetsolvecallbackfunc (CPXENVptr env,
                             int (CPXPUBLIC *solvecallback)
                                  (CPXCENVptr env,
                                   void *cbdata,
                                   int wherefrom,
                                   void *cbhandle,
                                   int *useraction_p),
                             void *cbhandle);
```

Arguments

`CPXENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`int (CPXPUBLIC *solvecallback)`

A pointer to a user-written solve callback. If the callback is set to `NULL`, no callback is called during optimization.

`void *cbhandle`

A pointer to user private data. This pointer is passed to the callback.

Callback

Description ILOG CPLEX calls the solve callback before ILOG CPLEX solves the subproblem defined by the current node. The user can choose to solve the subproblem in the solve callback instead by setting the user action parameter of the callback. The optimization that the user provides to solve the subproblem must provide a ILOG CPLEX solution. That is, the ILOG CPLEX Callable Library routine `CPXgetstat()`, documented in the *ILOG CPLEX Reference Manual*, must return a nonzero value. The user may access the lp pointer of the subproblem with the Callable Library routine `CPXgetcallbacknodelp()`.

Return Value The callback returns a zero on success, and a nonzero if an error occurs.

Arguments `CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`void *cbdata`

A pointer passed from the optimization routine to the user-written callback that identifies the problem being optimized. The only purpose of this pointer is to pass it to the callback information routines.

`int wherefrom`

An integer value indicating where in the optimization this function was called. It will have the value `CPX_CALLBACK_MIP_SOLVE`.

`void *cbhandle`

A pointer to user private data.

`int *useraction_p`

A pointer to an integer indicating the to be taken on completion of the user callback. Table 11 summarizes possible actions.

Table 11 *Actions to be Taken after a User-Written Solve Callback*

Value	Symbolic Constant	Action
0	CPX_CALLBACK_DEFAULT	Use ILOG CPLEX subproblem optimizer
1	CPX_CALLBACK_FAIL	Exit optimization
2	CPX_CALLBACK_SET	The subproblem has been solved in the callback

Example

```
status = CPXsetsolvecallbackfunc(env, mysolvefunc, mydata);
```

See Also

The example `admipex1.c`

CPXgetsolvecallbackfunc(), CPXgetcallbacknodelp(), Advanced MIP Control Interface

CPXslackfromx

Usage Advanced

Description The routine `CPXslackfromx()` computes an array of slack values from primal solution values.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXslackfromx (CPXCENVptr env,
                   CPXCLPptr lp,
                   const double *x,
                   double *slack);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`const double *x`

An array that contains primal solution (`x`) values for the problem, as returned by routines such as `CPXcrushx()` and `CPXuncrushx()`. The array must be of length at least the number of columns in the LP problem object.

`double *slack`

An array to receive the slack values computed from the x values for the problem object. The array must be of length at least the number of rows in the LP problem object.

Example

```
status = CPXslackfromx (env, lp, x, slack);
```

CPXsolwrite

Usage Advanced

Description The routine `CPXsolwrite()` is a generic routine for writing solutions. It performs all the calculations needed to produce a solution file, but it writes only through functions that the user provides to it, so that the user may choose the data representation, data selection, and file format.

The user must open the file before calling `CPXsolwrite()` and close the file after calling `CPXsolwrite`().

The arguments to `CPXsolwrite()` are functions it calls to write the file. `CPXsolwrite()` does not "know" anything about the file or the type of output being written. The argument `info`, the last parameter, communicates information to the routines `hsection`, `rsectionbeg`, `csectionbeg`, `write_entry`, and `sectionend`.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXsolwrite (CPXENVptr env,
                 CPXLPptr lp,
                 void (CPXPUBLIC *hsection)(CPXENVptr env,
                                            CPXLPptr lp,
                                            void *info),
                 void (CPXPUBLIC *rsectionbeg) (void *info),
                 void (CPXPUBLIC *csectionbeg) (void *info),
                 void (CPXPUBLIC *write_entry) (void *info,
                                                int aflag,
                                                int num,
                                                char *name,
                                                char *state,
                                                double val1,
                                                double val2,
                                                double ll,
                                                double ul,
                                                double val3),
                 void (CPXPUBLIC *sectionend) (void *info),
                 void *info);
```

Arguments

`CPXENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

```
void (CPXPUBLIC *hsection)(CPXENVptr env,
                           CPXLPptr lp,
                           void *info)
```

The function `hsection` writes a header section in a formatted file. Its only arguments are the ILOG CPLEX environment pointer `env`, the problem pointer `lp`, and the `info` parameter. `CPXsolwrite()` calls this function first. It uses the `lp` problem pointer to retrieve any information needed by the header-section function.

```
void (CPXPUBLIC *rsectionbeg) (void *info)
```

The function `rsectionbeg` writes information at the beginning of the row section of a formatted file. It is called after `hsection` and `sectionend`. Its only argument is the `info` parameter.

```
void (CPXPUBLIC *csectionbeg) (void *info)
```

The function `csection` writes information at the beginning of the column section of a formatted file. It is called after all row entries have been completed. Its only argument is the `info` parameter.

```
void (CPXPUBLIC *write_entry) (void *info,
                               int aflag,
                               int num,
                               char *name,
                               char *state,
                               double val1,
                               double val2,
                               double ll,
                               double ul,
                               double val3)
```

The function `write_entry` is called once for each row and column in the problem. Table 12 summarizes its arguments.

Table 12 *Arguments of the `write_entry` Function in the Routine `CPXsolwrite()`*

Type	Name	Meaning
`void`	`*info`	the `info` parameter of `CPXsolwrite`
`int`	`aflag`	a for alternate optimum
`int`	`num`	sequence number; cumulative over rows and columns
`char`	`*name`	name of row or column
`char`	`*state`	state of row or column; one of: `UL`, `LL`, `BS`, `EQ`, `FR`, `**`
`double`	`val1`	for rows, row activity; for columns, column solution value
`double`	`val2`	for rows, slack activitiy; for columns, objective coefficient
`double`	`ll`	for rows, lower limit; for columns, lower bound
`double`	`ul`	for rows, upper limit; for columns, upper bound
`double`	`val3`	for rows, dual value; for columns, reduced cost

`void (CPXPUBLIC *sectionend) (void *info)`

The function `sectionend` is used at the end of each header, row, and column section. The only argument to this function is the `info` parameter.

`void *info`

A generic pointer that passes information to each of the functions called by `CPXsolwrite()`.

CPXstrongbranch

Usage

Advanced

Description

The routine `CPXstrongbranch()` computes information for selecting a branching variable in an integer-programming branch & cut search.

To describe this routine, let's assume that an LP has been solved and that the optimal solution is resident. Let `goodlist[]` be the list of variable indices for this problem and `goodlen` be the length of that list. Then `goodlist[]` gives rise to `2*goodlen` different LPs in which each of the listed variables in turn is fixed to the greatest integer value less than or equal to its value in the current optimal solution, and then each variable is fixed to the least integer value greater than or equal to its value in the current optimal solution. `CPXstrongbranch` performs at most `itlim` dual steepest-edge iterations on each of these `2*goodlen` LPs, starting from the current optimal solution of the base LP. The values that these iterations yield are placed in the arrays `downpen[]` for the downward fix and `uppen[]` for the upward fix. Setting `CPX_PARAM_DGRADIENT` to 2 may give more informative values for the arguments `downpen[]` and `uppen[]` for a given number of iterations `itlim`.

A user might use other routines of the ILOG CPLEX Callable Library directly to build a function that computes the same values as `CPXstrongbranch()`. However, `CPXstrongbranch()` should be faster because it takes advantage of direct access to internal ILOG CPLEX data structures.

Return Value

The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXstrongbranch (CPXCENVptr env,
                     CPXLPptr lp,
                     const int *goodlist,
                     int goodlen,
                     double *downpen,
                     double *uppen,
                     int itlim);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`const int *goodlist`

An array of integers. The length of the array must be at least `goodlen`. As in other ILOG CPLEX Callable Library routines, row variables in `goodlist[]` are specified by the negative of row index shifted down by one; that is, `-rowindex -1`.

`int goodlen`

An integer indicating the number of entries in `goodlist[]`.

`double *downpen`

An array containing values that are the result of the downward fix of branching variables in dual steepest-edge iterations carried out by `CPXstrongbranch()`. The length of the array must be at least `goodlen`.

`double *uppen`

An array containing values that are the result of the upward fix of branching variables in dual steepest-edge iterations carried out by `CPXstrongbranch()`. The length of the array must be at least `goodlen`.

`int itlim`

An integer indicating the limit on the number of dual steepest-edge iterations carried out by `CPXstrongbranch()` on each LP.

CPXtightenbds

Usage Advanced

Description The routine `CPXtightenbds()` changes the upper or lower bounds on a set of variables in a problem. Several bounds can be changed at once. Each bound is specified by the index of the variable associated with it. The value of a variable can be fixed at one value by setting both the upper and lower bounds to the same value.

In contrast to the ILOG CPLEX Callable Library routine `CPXchgbds()`, also used to change bounds, `CPXtightenbds()`preserves more of the internal ILOG CPLEX data structures so it is more efficient for re-optimization, particularly when changes are made to bounds on basic variables.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXtightenbds (CPXCENVptr env,
                   CPXLPptr lp,
                   int cnt,
                   const int *indices,
                   const char *lu,
                   const double *bd);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int cnt`

An integer indicating the total number of bounds to change. That is, `cnt` specifies the length of the arrays `indices`, `lu`, and `bd`.

`const int *indices`

An array containing the numerical indices of the columns corresponding to the variables for which bounds will be changed. The allocated length of the array is `cnt`. Column *j* of the constraint matrix has the internal index `j - 1`.

`const char *lu`

An array containing characters indicating whether the corresponding entry in the array `bd` specifies the lower or upper bound on column `indices[j]`. The allocated length of the array is `cnt`. Table 13 summarizes the values that entries in this array may assume.

Table 13 *Bound Indicators in the Argument* `lu` *of* `CPXtightenbds()`

Value of lu[j]	Meaning for bd[j]
`U`	`bd[j]` is an upper bound
`L`	`bd[j]` is a lower bound
`B`	`bd[j]` is the lower and upper bound

`const double *bd`

An array containing the new values of the upper or lower bounds of the variables present in the array `indices`. The allocated length of the array is `cnt`.

Example

```
status = CPXtightenbds (env, lp, cnt, indices, lu, bd);
```

CPXuncrushform

Usage Advanced

Description The routine `CPXuncrushform()` uncrushes a linear formula of the presolved problem to a linear formula of the original problem.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXuncrushform (CPXCENVptr env,
                    CPXCLPptr lp,
                    int plen,
                    const int *pind,
                    const double *pval,
                    int *len_p,
                    double *offset_p,
                    int *ind,
                    double *val);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`int plen`

The number of entries in the arrays `pind` and `pval`.

`const int *pind`
`const double *pval`

The linear formula in terms of the presolved problem. Each entry, `pind[i]`, indicates the column index of the corresponding coefficient, `pval[i]`.

`int *len_p`

A pointer to an integer to receive the number of nonzero coefficients, that is, the true length of the arrays `ind` and `val`.

`double *offset_p`

A pointer to a double to contain the value of the linear formula corresponding to variables that have been removed in the presolved problem.

```
int *ind
double *val
```

The linear formula in terms of the original problem.

Let `cols = CPXgetnumcols (env, lp)`. If `ind[i] < cols` then the $\mathtt{i}^{th}$ variable in the formula is variable with index `ind[i]` in the original problem. If `ind[i] >= cols`, then the $\mathtt{i}^{th}$ variable in the formula is the slack for the (`ind[i] - cols`)th ranged row. The arrays `ind` and `val` must be of length at least the number of columns plus the number of ranged rows in the original LP problem object.

Example

```
status = CPXuncrushform (env, lp, plen, pind, pval,
                         &len, &offset, ind, val);
```

CPXuncrushpi

Usage Advanced

Description The routine `CPXuncrushpi()` uncrushes a dual solution for the presolved problem to a dual solution for the original problem. This routine is for linear programs. Use `CPXqpuncrushpi()` for quadratic programs.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXuncrushpi (CPXCENVptr env,
                  CPXCLPptr lp,
                  double *pi,
                  const double *prepi);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`double *pi`

An array to receive dual solution (`pi`) values for the original problem as computed from the dual values of the presolved problem object. The array must be of length at least the number of rows in the LP problem object.

`const double *prepi`

An array that contains dual solution (`pi`) values for the presolved problem, as returned by routines such as `CPXgetpi()` and `CPXsolution()` when applied to the presolved problem object. The array must be of length at least the number of rows in the presolved problem object.

Example

```
status = CPXuncrushpi (env, lp, pi, prepi);
```

CPXuncrushx

Usage Advanced

Description The routine `CPXuncrushx()` uncrushes a solution for the presolved problem to the solution for the original problem.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXuncrushx (CPXCENVptr env,
                 CPXCLPptr lp,
                 double *x,
                 double *prex);
```

Arguments

`CPXCENVptr env`

The pointer to the CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXCLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

`double *x`

An array to receive the primal solution (`x`) values for the original problem as computed from primal values of the presolved problem object. The array must be of length at least the number of columns in the LP problem object.

`double *prex`

An array that contains primal solution (`x`) values for the presolved problem, as returned by routines such as `CPXgetx()` and `CPXsolution()` when applied to the presolved problem object. The array must be of length at least the number of columns in the presolved problem object.

Example

```
status = CPXuncrushx (env, lp, x, prex);
```

CPXunscaleprob

Usage Advanced

Description The routine `CPXunscaleprob()` removes any scaling that ILOG CPLEX has applied to the resident problem and its associated data. A side effect is that if there is a resident solution, any associated factorization is discarded and the solution itself is *deactivated*, meaning that it can no longer be accessed with a call to `CPXsolution()`, nor by any other query routine. However, any starting point information for the current solution (such as an associated basis) is retained.

Return Value The routine returns a zero on success, and a nonzero if an error occurs.

Synopsis

```
int CPXunscaleprob (CPXCENVptr env,
                    CPXLPptr lp);
```

Arguments

`CPXCENVptr env`

The pointer to the ILOG CPLEX environment, as returned by one of the `CPXopenCPLEX` routines.

`CPXLPptr lp`

A pointer to a CPLEX LP problem object, as returned by `CPXcreateprob`, documented in the *CPLEX Reference Manual*.

IloCplex

Category `Handle Class`

Inheritance Path `IloAlgorithm`
➥ **`IloCplex`**

Description We include here additional methods of `IloCplex`. For a description of the `IloCplex` class, see the *ILOG CPLEX Reference Manual*.

Member Functions

`IloRange` **`addLazyConstraint`**`(IloRange rng);`

This member function adds `rng` as a lazy constraint to the invoking `IloCplex` object. The range `rng` is copied into the lazy constraint pool; the `rng` itself is not part of the pool, so changes to `rng` after it has been copied into the lazy constraint pool will not affect the lazy constraint pool.

Lazy constraints added with `addLazyConstraint` are typically constraints of the model that are not expected to be violated when left out. The idea behind this is that the LPs that are solved when solving the MIP can be kept smaller when these constraints are not included. `IloCplex` will, however, include a lazy constraint in the LP as soon as it becomes violated. In other words, the solution computed by `IloCplex` ensures that all the lazy constraints that have been added are satisfied.

By contrast, if the constraint does not change the feasible region of the extracted model but only strengthens the formulation, it is referred to as a user cut. While user cuts can be added to `IloCplex` with `addLazyConstraint`, it is generally preferable to do so using `addUserCuts`. It is an error, however, to add lazy constraints using `addUserCuts`.

When columns are deleted from the extracted model, all lazy constraints are deleted as well and need to be recopied into the lazy constraint pool. Use of this method in place of `addCuts` allows for further presolve reductions

`IloRangeArray` **`addLazyConstraints`** `(IloRangeArray rng);`

This member function adds a set of lazy constraints to the invoking `IloCplex` object. Everything said for `addLazyConstraint` applies to each of the lazy constraints given in array `rng`.

`IloRange` **`addUserCut`**`(IloRange rng);`

This member function adds `rng` as a user cut to the invoking `IloCplex` object. The range `rng` is copied into the user cut pool; the `rng` itself is not part of the pool, so changes to `rng` after it has been copied into the user cut pool will not affect the user cut pool.

Cuts added with `addUserCut` must be real cuts, in that the solution of a MIP does not depend on whether the cuts are added or not. Instead, they are there only to strengthen the formulation.

> ***Note:*** *It is an error to use* `addUserCut` *for lazy constraints, that is, constraints whose absence may potentially change the solution of the problem. Use* `addLazyConstraints` *or, equivalently,* `addCut` *when adding such a constraint.*

When columns are deleted from the extracted model, all user cuts are deleted as well and need to be recopied into the user cut pool. This method is equivalent to `addCuts`, documented in the *ILOG CPLEX Reference Manual*.

```
IloRangeArray addUserCuts(IloRangeArray rng);
```

This member function adds a set of user cuts to the invoking `IloCplex` object. Everything said for `addUserCut` applies to each of the cuts given in array `rng`.

```
void basicPresolve(const IloNumVarArray vars,
                   IloNumArray          redlb = 0,
                   IloNumArray          redub = 0,
                   const IloRangeArray  rngs = 0,
                   IloBoolArray         redundant = 0) const;
```

This method can be used to compute tighter bounds for the model variables and to detect redundant constraints in the model extracted to the invoking `IloCplex` object. For every variable specified in parameter `vars`, it will return possibly tightened bounds in the corresponding elements of arrays `redlb` and `redub`. Similarly, for every constraint specified in parameter `rngs`, this method will return a boolean indicating whether or not it is redundant in the model in the corresponding element of array `redundant`.

```
void clearLazyConstraints();
```

This member function deletes all lazy constraints added to the invoking `IloCplex` object with the methods `addLazyConstraint` and `addLazyConstraints`.

```
void clearUserCuts();
```

This member function deletes all user cuts added to the invoking `IloCplex` object with the methods `addUserCut` and `addUserCuts`.

```
void freePresolve();
```

This member function frees the presolved problem. Under the default setting of parameter `Reduce`, the presolved problem is freed when an optimal solution is found; however, it is not freed if `Reduce` has been set to 1 (primal reductions) or to 2 (dual reductions). In these instances, the function `freePresolve()` can be used when necessary to free it manually.

```
IloExtractable getDiverging();
```

This member function returns the diverging variable or constraint, in a case where the primal Simplex algorithm has determined the problem to be infeasible. The returned extractable is either an `IloNumVar` or an `IloRange` object extracted to the invoking `IloCplex` optimizer; it is of type `IloNumVar` if the diverging column corresponds to a variable, or of type `IloRange` if the diverging column corresponds to the slack variable of a constraint.

```
void getRay();
```

This member function returns an unbounded direction corresponding to the present basis for an LP that has been determined to be an unbounded problem.

```
void  importModel(IloModel& m,
                 const char* filename,
                 IloObjective& obj,
                 IloNumVarArray vars,
                 IloRangeArray  rngs,
                 IloRangeArray  lazy = 0,
                 IloRangeArray  cuts = 0) const;
```

This method is an extension of the `importModel` method documented in the Reference Manual but has two extra parameters, `lazy` and `cuts`. If a non-zero array handle is passed as parameter `lazy`, it will be filled with `IloRange` objects corresponding to all lazy constraints that are found in the model file. Similarly, if a non-zero array handle is passed as parameter `cuts`, it will be filled with `IloRange` objects corresponding to all user cuts that are found in the model file. The rest of the parameters are exactly the same as the parameters of the corresponding `importModel` method documented in the *ILOG CPLEX Reference Manual*.

```
void importModel(IloModel& m,
                 const char* filename,
                 IloObjective& obj,
                 IloNumVarArray vars,
                 IloRangeArray  rngs,
                 IloSOS1Array sos1,
                 IloSOS2Array sos2),
                 IloRangeArray  lazy = 0,
                 IloRangeArray  cuts = 0) const;
```

This method is an extension of the `importModel` method documented in the Reference Manual but has two extra parameters, `lazy` and `cuts`. If a non-zero array handle is passed as parameter `lazy`, it will be filled with `IloRange` objects corresponding to all lazy constraints that are found in the model file. Similarly, if a non-zero array handle is passed as parameter `cuts`, it will be filled with `IloRange` objects corresponding to all user cuts that are found in the model file. The rest of the parameters are exactly the same as the parameters of the corresponding `importModel` method documented in the *ILOG CPLEX Reference Manual*.

```
void presolve(IloCplex::Algorithm alg);
```

This member function performs Presolve on the model. The enumeration `alg` tells Presolve which algorithm is intended to be used on the reduced model; `NoAlg` should be specified for MIP models.

IloCplex::LazyConstraintCallbackI

Category Nested Class

Inheritance Path

```
IloCplex::CallbackI
➥ IloCplex::MIPCallbackI
 ➥ IloCplex::ControlCallbackI
  ➥ IloCplex::CutCallbackI
    ➥ IloCplex::LazyConstraintCallbackI
```

Description An instance of the class `IloCplex::LazyConstraintCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a MIP while generating lazy constraints. `IloCplex` calls the user-written callback after solving each node LP exactly like `IloCplex::CutCallbackI`. In fact, this callback is exactly equivalent to `IloCplex::CutCallbackI` but offers a name more consistently pointing out the difference between lazy constraints and user cuts.

Include File `<ilcplex/ilocplex.h>`

Definition File `<ilcplex/ilocplexi.h>`

Synopsis

```
class LazyConstraintCallbackI : public CutCallbackI {
  protected :
  LazyConstraintCallbackI()
};
```

IloCplex::UserCutCallbackI

Category Nested Class

Inheritance Path

`IloCplex::CallbackI`
➥ `IloCplex::MIPCallbackI`
➥ `IloCplex::ControlCallbackI`
➥ `IloCplex::CutCallbackI`
➥ **`IloCplex::UserCutCallbackI`**

Description An instance of the class `IloCplex::UserCutCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a MIP while generating user cuts to tighten the LP relaxation. `IloCplex` calls the user-written callback after solving each node LP exactly like `IloCplex::CutCallbackI`. The only difference to `IloCplex::CutCallbackI` is that constraints added in a `UserCutCallbackI` must be real cuts in the sense that omitting them does not affect the feasible region of the model under consideration.

Include File `<ilcplex/ilocplex.h>`

Definition File `<ilcplex/ilocplexi.h>`

Synopsis

```
class UserCutCallbackI : public CutCallbackI {
  protected :
  UserCutCallbackI()
};
```

ILOLAZYCONSTRAINTCALLBACK

Category Macro

Synopsis

```
ILOLAZYCONSTRAINTCALLBACK0(name)
ILOLAZYCONSTRAINTCALLBACK1(name, type1, x1)
ILOLAZYCONSTRAINTCALLBACK2(name, type1, x1, type2, x2)
ILOLAZYCONSTRAINTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOLAZYCONSTRAINTCALLBACK4(name, type1, x1, type2, x2, type3, x3,
                           type4, x4)
ILOLAZYCONSTRAINTCALLBACK5(name, type1, x1, type2, x2, type3, x3,
                           type4, x4, type5, x5)
ILOLAZYCONSTRAINTCALLBACK6(name, type1, x1, type2, x2, type3, x3,
                           type4, x4, type5, x5, type6, x6)
ILOLAZYCONSTRAINTCALLBACK7(name, type1, x1, type2, x2, type3, x3,
                           type4, x4, type5, x5, type6, x6,
                           type 7, x7)
```

Description

This macro creates two things, an implementation class for a user-defined lazy constraint callback named `nameI` and a function named `name()` that creates an instance of this class and returns an `IloCplex::Callback` handle for it. This function needs to be called with an environment as first parameter followed by the *n* parameters specified at the macro execution in order to create a callback. The callback can then be used, passing it the `use()` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of method `makeClone()` as required for callbacks. The implementation of method `main()` must be provided by the user in parentheses {} following the macro invocation:

```
ILOLAZYCONSTRAINTCALLBACKn(name, ...) {
// implementation of the callback
}
```

For the implementation of the callback, methods from class `IloCplex::LazyConstraintCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

Include File `<ilcplex/ilocplex.h>`

Definition File `<ilcplex/ilocplex.h>`

See Also *IloCplex::LazyConstraintCallbackI*

ILOUSERCUTCALLBACK

Category Macro

Synopsis

```
ILOUSERCUTCALLBACK0(name)
ILOUSERCUTCALLBACK1(name, type1, x1)
ILOUSERCUTCALLBACK2(name, type1, x1, type2, x2)
ILOUSERCUTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOUSERCUTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4,
                    x4)
ILOUSERCUTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4,
                    x4, type5, x5)
ILOUSERCUTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4,
                    x4, type5, x5, type6, x6)
ILOUSERCUTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4,
                    x4, type5, x5, type6, x6, type 7, x7)
```

Description This macro creates two things, an implementation class for a user-defined user cut callback named `nameI` and a function named `name()` that creates an instance of this class and returns an `IloCplex::Callback` handle for it. This function needs to be called with an environment as first parameter followed by the *n* parameters specified at the macro execution in order to create a callback. The callback can then be used, passing it the `use()` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of method `makeClone()` as required for callbacks. The implementation of method `main()` must be provided by the user in parentheses {} following the macro invocation:

```
ILOUSERCUTCALLBACKn(name, ...) {
// implementation of the callback
}
```

For the implementation of the callback, methods from class `IloCplex::UserCutCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

Include File `<ilcplex/ilocplex.h>`

Definition File `<ilcplex/ilocplex.h>`

See Also *IloCplex::UserCutCallbackI*

A

Advanced Features Release Notes

This appendix includes:

- CPLEX 8.1 Advanced Features Release Notes
- CPLEX 8.0 Advanced Features Release Notes
- CPLEX 7.1 Advanced Features Release Notes
- CPLEX 6.6 to 7.0 Advanced Features Release Notes

CPLEX 8.1 Advanced Features Release Notes

No conversion steps pertaining to advanced features are necessary in order to move from CPLEX 8.0 to CPLEX 8.1.

This is also the case with the standard features of CPLEX, as documented in the CPLEX 8.1 Release Notes.

CPLEX 8.0 Advanced Features Release Notes

ILOG CPLEX 8.0 offers a number of new routines and extensions to existing routines in the Callable Library. There are no advanced feature changes for Concert Technology.

Conversion Notes

The introduction of `const` arguments throughout the CPLEX Callable Library requires that user-written callback routines be modified to use `const` arguments.

The function used to specify branching within a branch callback has been changed. The function has been renamed from `CPXbranchcallbackbranch()` to `CPXbranchcallbackbranchbds()` to reflect that this is the particular branching function that creates a node by changing variable bounds. A parameter has been added to allow the association of user data with the created node.

The function `CPXgetkappa()` has been changed to return an estimate of the condition number of simplex basis instead of the exact value. The estimate can be computed more quickly than the exact condition number. A new function, `CPXgetExactkappa()`, returns the exact condition number as `CPXgetkappa()` did in previous releases.

New features

Exact Condition Number

As noted in the Conversion Notes, the existing Callable Library routine `CPXgetkappa()` now returns an upper bound on the condition number of the simplex basis. The exact condition number can be obtained by calling the new function `CPXgetExactkappa()`.

Branching on Constraints

Two new branching functions are provided for branching on constraints from the branch callback. They are `CPXbranchcallbackbranchconstraints()`, where the branch is created by adding one or more constraints to a problem, and `CPXbranchcallbackbranchgeneral()`, where the branch is created by adding one or more constraints and changing one or more variable bounds.

Local Cuts

The function `CPXcutcallbackaddlocal()` is called from the cut callback to add cuts that apply to the subtree rooted at the current node. This contrasts with the function `CPXcutcallbackadd()` which is used to add global cuts, that is, cuts that are valid at all nodes.

Associating User Data with Nodes

The delete node callback is called when a node is removed from the branch & bound tree, either by branching on it or by fathoming it. It can be used to free user data that was associated with a node. The routine `CPXsetdeletenodecallbackfunc()` instructs CPLEX to use the specified user-written callback routine whenever a node is deleted. The routine `CPXsetdeletenodecallbackfunc()` accesses the user-written routine currently being used.

Obtaining Node Information by Sequence Number

The new routine `CPXgetcallbackseqinfo()` is used to query branch & cut nodes during the node callback. The node sequence number is used to identify the node instead of the node index number.

Information on the Current Node

The routine `CPXgetcallbacknodeinfo()` can be used in the branch, incumbent, and heuristic callbacks to obtain information on the current node.

Advanced Presolve Functions for QP

Many of the advanced presolve routines have been extended for QP, such as `CPXcrushx()` and `CPXuncrushx()`. However, two new routines are needed for transforming dual values between the original and presolved QP problems: `CPXqpuncrushpi()` and `CPXqpdjfrompi()`.

CPLEX 7.1 Advanced Features Release Notes

CPLEX 7.1 offers a number of new routines and extensions to existing routines, both in the Callable Library and Concert Technology Library.

Conversion Notes

The following routines have been modified.

- The branch callback function has been changed so that additional information is available to the user and so that users may branch from integer feasible nodes. Previously, the branches to be created in a branch callback were specified by filling the arguments to the callback function; in 7.1 a new function, `CPXbranchcallbackbranchbds()`, is called to specify the branches. The node sequence number assigned to the node created by the specified branch is returned. See the sections for:
  - `CPXsetbranchcallbackfunc()`
  - `CPXbranchcallbackbranchbds()`
- In the Concert Technology Library, `IloCplex::importModel` methods are extended, with extra parameters for user cuts and lazy constraints.

New Features

Incumbent Callback

A new callback, the incumbent callback, has been added so that users may examine integer feasible solutions to decide if they should replace the current incumbent. This callback allows discarding solutions that do not meet criteria additional to that expressed in the mixed integer program formulation.

The incumbent callback, together with the additional calls to the branch callback for integer feasible nodes, provide a means to specify logical constraints instead of introducing indicator variables. For example, the user could accept only solutions where at least two variables from a set of continuous variables must be nonzero. Solutions not meeting this criteria could be rejected in an incumbent callback, and additional branches changing the bounds on these variables to nonzero values could be specified by a branch callback.

See the new routines:

- `CPXsetincumbentcallbackfunc()`
- `CPXgetincumbentcallbackfunc()`

User Cuts

In the Concert Technology Library, the following are added to provide capabilities for user cuts:

- `IloCplex::addUserCut`
- `IloCplex::addUserCuts`
- `IloCplex::clearUserCuts`
- `IloCplex::UserCutCallbackI`
- `ILOUSERCUTCALLBACK` macro

Lazy Constraints

Two new functions have been added for the Callable Library:

- `CPXaddlazyconstraints()`
- `CPXfreelazyconstraints()`

They are used for constraints that are not implied by the constraints in the MIP problem. In the prior version 7.0, these constraints could be specified by `CPXaddusercuts()`, but in 7.1 `CPXaddusercuts()` should be used only to specify cuts, that is, additional constraints that *are* implied by the constraints in the MIP formulation but that may be helpful in obtaining a proved optimal integer solution. `CPXaddlazyconstraints()` and `CPXfreelazyconstraints()` have the same arguments as the user cut functions.

In the Concert Technology Library, the following are added to provide capabilities for lazy constraints:

- `IloCplex::addLazyConstraint`
- `IloCplex::addLazyConstraints`
- `IloCplex::clearLazyConstraints`
- `IloCplex::LazyConstraintCallbackI`
- `ILOLAZYCONSTRAINTCALLBACK` macro

`CPXmipopt()` will return with an error when the advanced presolve parameters are set in conflict with the presence of user cuts and/or lazy constraints. For lazy constraints, `CPX_PARAM_REDUCE` must be set to `CPX_PREREDUCE_NOPRIMALORDUAL` or `CPX_PREREDUCE_PRIMALONLY`. For user cuts, `CPX_PARAM_REDUCE` must be set to `CPX_PREREDUCE_NOPRIMALORDUAL` or `CPX_PREREDUCE_PRIMALONLY`; or `CPX_PARAM_PRELINEAR` must be set to `1`.

A new chapter has been added to provide information on using the user cuts and lazy restraints. See Chapter 3, *User Cut and Lazy Constraint Pools*.

CPLEX 6.6 to 7.0 Advanced Features Release Notes

Conversion Notes

The following routines replace the routine `CPXgetvarcallbackinfo`:

- `CPXgetcallbacknodex`
- `CPXgetcallbacknodeobjval`
- `CPXgetcallbackctype`
- `CPXgetcallbackorder`
- `CPXgetcallbackpseudocosts`
- `CPXgetcallbackincumbent`
- `CPXgetcallbacknodeintfeas`
- `CPXgetcallbackgloballb`
- `CPXgetcallbackglobalub`
- `CPXgetcallbacknodelb`
- `CPXgetcallbacknodeub`
- `CPXgetcallbacknodestat`
- `CPXgetcallbacklp`

The following routines have been renamed:

- `CPXgetnodecallbackinfo` has been renamed CPXgetcallbacknodeinfo.
- `CPXgetsoscallbackinfo` has been renamed CPXgetcallbacksosinfo.
- `CPXgetsubcallbackinfo` has been renamed CPXgetcallbacknodelp.

New Features

Advanced Presolve

An Advanced Presolve Interface has been added. It includes features for controlling presolve, access to the presolve problem, and accessing original variables from within MIP control callbacks.

CPXpivot Routine

A routine to specify a simplex pivot, `CPXpivot()`, has been added.

CPXgetray Routine

A routine to return an unbounded direction vector for primal unbounded or dual infeasible problems has been added: `CPXgetray()`.

List of Tables

Index

C (continued)

D

F

G

H

Ilo

I (continued)

K

L

M

W